UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE FÍSICA

Functional Tester for High Voltage Boards of the TILECAL Calorimeter

Filipe Mendes de Oliveira Cuim

Mestrado Integrado em Engenharia Física

Dissertação orientada por:
Prof. Guiomar Gaspar de Andrade Evans
Prof. José António Soares Augusto

2019

Acknowledgements

This work is dedicated to my family that gave me excellent conditions to continue my studies and always encouraged me to take full advantage of my years in the university.

I also want to thank all the professors of the Faculty of Sciences that helped me during my academic years be it for personal or scholar matters.

To the Department of Physics of the Faculty of Sciences of the University of Lisbon that gave me the space and material to help develop my work.

A big thanks to professor José Soares Augusto, professor Guiomar Evans and Filipe Martins, that followed my work and gave me a tremendous support during its development.

Finally, to my friends that have also supported and helped me and with whom I shared one of the greatest experiences of my life.

Abstract

This work was done in the scope of a project, called HV Remote, with its main goal being to build an improved high voltage regulation system for the ATLAS TileCal's photomultiplier tubes (PMTs). This project consists on a remote system that comprises a HV distribution system, called HV Remote board, a high voltage and low voltage supplies board, called Power Supplies board, and several multi-conductor cables that distribute the high voltage to the PMTs. The Power supplies board provides the LV (low voltage) that the HV Remote needs to work and feeds a fixed HV (high voltage) value to it. The HV Remote individually regulates the HV provided by the Power supplies board and monitors the PMTs' high voltage, being also able to disable any of them in the case of a malfunction.

The TileCal has approximately 10000 PMTs, so this new system has 256 HV Remote and dedicated supplies boards, where each pair of boards is associated to 48 PMTs. When these boards are produced, after the design phase, they will need to be the target of several functional tests, and high-temperature tests. This work focuses on designing a software interface for the HV Remote boards, that will be incorporated, in the future, in the DCS (Detector Control System) of ATLAS and a GUI (Graphical User Interface) to perform all the necessary tests to the boards, before they are sent to CERN.

Each group of 16 boards will be controlled by a FPGA, but in this work a Raspberry Pi will be used as the master controller for testing the control system architecture. The Raspberry Pi will use the standard Serial Peripheral Interface (SPI) protocol to establish the communication with the boards. The code of the software interface will be written with Python 3 and will be specifically designed for the Raspberry Pi. So, in the future, the code must be adapted to the FPGA. The GUI for the tests will be designed using the PyQt5 library of Python. At present, and during the development of this thesis' work, the boards weren't produced yet, so some of its main digital components (port expanders, DACs, ADC and MUXs) were bought and assembled, in order to test the code, the GUI and the overall communication interface of the Raspberry Pi. These components were assembled in a breadboard, and the connections mimicked the HV Remote board's schematics.

Keywords: TileCal, HV Remote, Raspberry Pi, Python, SPI.

Resumo

Este trabalho foi realizado no âmbito de um projeto de desenvolvimento de um sistema remoto de distribuição e controlo da alta tensão fornecida aos fotomultiplicadores usados no TileCal, na experiência ATLAS no CERN. O sistema consiste num conjunto de 256 pares de cartas eletrónicas: uma para controlo e distribuição das altas tensões individuais aos PMTs, denominada HV Remote, e outra de alimentação, denominada de Power Supplies. A carta de alimentação fornece um de dois possíveis valores de alta tensão para a HV Remote, sendo eles -830 V ou -950 V. Para além disso ela fornece também a baixa tensão que a HV Remote necessita para funcionar, que são ± 12 V para os componentes analógicos e 3.3 V para os componentes digitais, bem como as suas respetivas massas. A carta HV Remote tem as seguintes funções:

- Regular individualmente o valor de alta tensão fornecido aos fotomultiplicadores, sendo isto possível com um conjunto de conversores digitais-analógicos (DACs);
- Monitorizar as suas altas tensões, realizando leituras de tensão nos seus canais com um conjunto de multiplexadores (MUXs) e um conversor analógico-digital (ADC);
- Ativar/desativar os canais dos fotomultiplicadores, para o caso de haver alguma avaria. Isto permite que as medições realizadas nos fotomultiplicadores funcionais não sejam perturbadas pelos que apresentam um mau funcionamento;
- Realizar leituras de sensores de temperatura, que permitem ter um melhor controlo do aumento de temperatura das placas em regiões mais densamente povoadas por componentes.

As 256 cartas serão agrupadas em grupos de 16, sendo cada grupo controlado por uma FPGA no sistema final, quando as cartas forem enviadas para o CERN. Neste momento as cartas estão em fase de produção, e posteriormente terão de ser alvo de testes funcionais e testes de temperatura. Nesta fase, um Raspberry Pi (o modelo usado é o 3b+), substitui as FPGA. O Raspberry Pi (e no futuro a FPGA) controlará as cartas, estabelecendo um protocolo de comunicação SPI, que é uma interface de comunicação em série baseada uma arquitetura mestre-escravo, em que o dispositivo mestre (o Raspberry Pi) controla toda a comunicação, que envolverá transferências de palavras de 8 bits. Como o controlador tem de comunicar com vários componentes, e o protocolo de comunicação escolhido é do tipo série, expansores série-paralelo ("port expanders") foram usados para converter as palavras digitais enviadas em série, para 16 portas de saída que permitem a comunicação com todos os circuitos integrados. Alguns dos expansores foram também usados para implementar a funcionalidade de ativação/desativação dos canais dos fotomultiplicadores.

O protocolo SPI assenta em 3 sinais principais: MOSI ("Master Output Slave Input" ou saída do mestre e entrada do escravo), MISO ("Master Input Slave Output" ou entrada do mestre e saída do escravo) e SCLK (sinal de relógio). Adicionalmente um sinal, geralmente representado pela sigla CS ("Chip Select" – selecionador de chip) ou SS ("Slave Select" – selecionar de dispositivo escravo), é usado para ativar a comunicação com os componentes digitais. Na carta HV Remote cada componente é associado a um sinal CS e os três sinais principais do SPI serão partilhados por todos eles. O Raspberry poderá posteriormente comunicar com um computador através de uma ligação ethernet, sendo assim controlado remotamente ou poderá ser ele próprio usado como um computador, conectando-o a um ecrã. Este trabalho foi sempre desenvolvido diretamente no Raspberry Pi, montando-o como um computador convencional.

Este trabalho é sobre o desenvolvimento do software que permitirá a comunicação entre o Raspberry e a HV Remote e no desenvolvimento de uma interface gráfica (GUI) que será executada no Raspberry Pi para testar as placas HV Remote quando forem produzidas. O

programa de controlo foi escrito usando a linguagem Python 3, seguindo o formalismo da programação orientada para objetos, que envolve estruturas de classes constituídas por um número de métodos/funções, que permitem definir e emular objetos numa linguagem de programação. Por isso, foram criadas classes que representam cada um dos componentes que participam no controlo digital da HV Remote, sendo eles: o MCP23S17 (expansor série-paralelo), o DAC7568 (conversor digital-analógico) e o MAX1240 (conversor analógico-digital). Os multiplexadores não têm uma classe associada pois não têm nenhuma interface de comunicação, pois podem ser manuseados apenas pela alteração dos valores lógicos dos seus endereços (função que é realizada pelos expansores). Após a criação destas três classes, foi desenhada uma classe para a HV Remote, com os métodos necessários para realizar as funções das cartas, que incorpora e complementa as outras classes. As classes mais complicadas de criar foram a do expansor e do DAC, pois a classe do ADC contem apenas um método, devido ao facto de este não ter mais nenhuma funcionalidade para além da leitura da tensão na sua entrada. O DAC e o expansor têm diferentes funcionalidades que permitem realizar diferentes formas de escrita de tensões ou aceder de diferentes formas à referência interna, no caso do DAC, e configurar de diferentes formas as portas de saída, no caso do expansor.

Após a escrita das classes, começou-se a escrever o código da interface gráfica. Esta GUI foi realizada usando um módulo do Python, denominado PyQt5, que é um conjunto de "atalhos" que ligam a uma biblioteca escrita em C++ denominada Qt5. Este módulo foi escolhido, pois disponibiliza uma série de funcionalidades que no momento pareceram ser úteis para este projeto. O PyQt5 baseia-se num conjunto de objetos denominados "layouts" e "widgets". Os "widgets" são todas os objetos de interação que usualmente se observa em aplicações: botões, quadrados de selecionar ("checkboxes"), listas de selecção, retângulos de introdução de números ou palavras, etc… Os "layouts" são objetos que contêm os "widgets" e que permitem organizar as suas posições e os seus tamanhos. A janela da GUI foi desenhada com dois painéis, um para o ajuste de tensões e ativação/desativação de canais, e outro para a monitorização da alta tensão dos fotomultiplicadores.

Durante o período de realização deste trabalho, as placas ainda não tinham sido produzidas, e por isso, para testar o código e a comunicação com o Raspberry Pi, foram comprados os componentes acima mencionados. Estes foram montados numa placa de ligações (breadboard), e as ligações foram feitas de acordo com os esquemas elétricos do controlo digital da HV Remote. Estes testes tiveram como principal objetivo, verificar o funcionamento de todo o software. Alguns LEDs foram também conectados na saída dos expansores para simular a funcionalidade de ativação/desativação dos canais dos fotomultiplicadores. O software de comunicação foi terminado, e os testes realizados deram indicações de um bom funcionamento. A GUI tem a parte do ajuste de tensões e de ativação/desativação terminada, faltando desenvolver a parte da monitorização dos canais dos fotomultiplicadores. Nesta última parte, pretende-se inserir gráficos, para cada canal, que atualizam o valor da tensão em tempo real. Ficou também por realizar uma função para a leitura das tensões dos sensores de temperatura, cujos valores serão também mostrados num gráfico que será atualizado em tempo real.

Para além das funcionalidades que faltam implementar na interface gráfica, depois de serem realizados alguns testes funcionais às cartas HV Remote, o software terá de ser adaptado para funcionar numa FPGA.

Palavras-chave: TileCal, HV Remote, Raspberry Pi, Python, SPI.

Index

1 Introduction 1
1.1 Scope of this work 1
1.2 Objectives 1
1.3 Software implementation 1
1.4 State of the HV Remote board 2
1.5 Document structure 2
2 The Atlas Experiment 3
2.1 Overview of the ATLAS detector 3
2.2 The Magnet System 5
2.3 The Inner Detector 6
2.3.1 The Pixel Detector 7
2.3.2 The semiconductor tracker (SCT) 7
2.3.3 The Transition Radiation Tracker (TRT) 8
2.4 The Muon Spectrometer 8
2.5 Calorimeters 9
3 The TileCal Calorimeter 10
3.1 Overview of the calorimeter 10
3.2 The Mechanical Structure of TileCal 10
3.3 Optical Elements: the scintillating tiles, the wavelength shifting fibres and the photomultipliers. 11
3.4 The old HV distribution system of the PMTs: HV Opto and HV Micro boards 13
3.5 The High-Luminosity Upgrade of the LHC 15
3.5.1 The ATLAS upgrade for the HL-LHC 16
3.5.2 The upgrades to the Tile Calorimeter 16
4 The HV Remote system 18
4.1 General Description of the HV Remote System 18
4.2 The SPI protocol 23
4.3 The Digital Devices of the HV Remote 25
4.3.1 The port expander MCP23S17 25
4.3.2 The DAC7568 29
4.3.3 The ADC MAX1240 34
4.3.4 The bus switch SN74CB3Q3245 35
4.3.5 The MUX36S16 36
4.4 Functional description of the digital control of HV Remote 37
5 The HV Remote boards Tester 42

5.1 General Description of the tester 42
5.2 The Raspberry Pi 3b+ 43
5.3 The HV Remote software interface 47
5.3.1 The MCP23S17 class 47
5.3.2 The DAC7568 class 53
5.3.3 The MAX1240 class 56
5.3.4 The HV Remote class 58
5.4 The GUI and the breadboard assemble 60
5.5 Test of the DAC and the of the ADC connected in tandem 66
6 Conclusion 73
6.1 Summary of the activities and goals achieved 73
6.2 Future work 73
6.3 Personal reflection 74
7 References 75
8 Annex 77
8.1 Tables of all the DAC7568 commands 77
8.2 Port Expander's Connections 80
8.2.1 Port Expander A 80
8.2.2 Port Expander D 81
8.3 DAC1 connections 82
8.4 MUX1 Connections 83
8.5 Mapping of the PMTs channels 84
8.6 Code with the implementation of the HVREMOTE class 86
8.7 Code of the GUI 90
8.8 Code for testing the HVREMOTE class 95

Figures Index

Figure 1.1. Frontal picture of the HV Remote board on the left, and of the backward part on the right. 2
Figure 2.1. The ATLAS Detector layout. 4
Figure 2.2. Geometry of the magnetic system. In the centre is the solenoid system and around it, is the barrel toroid. At the sides are the end-cap toroids. 5
Figure 2.3. Image of the central Solenoid in the factory. 5
Figure 2.4. Picture of the Barrel Toroid. 5
Figure 2.5. Picture of one of the end-cap toroids. 5
Figure 2.6. A representation of the sub-detectors of the inner detector and their radial positions inside the cylindrical cavity. 6
Figure 2.7. Mechanical structure of the inner detector. 6
Figure 2.8. Scheme of the pixel detector. In the centre are the 3 pixel-layers, concentric to the beam axis and in the extremities are the 5 end-cap disk layers. 7
Figure 2.9. Layout of the semiconductor tracker (SCT). 7
Figure 2.10. Straw tubes mounted in one of the end-cap TRT wheels. 8
Figure 2.11. Layout of the muon chambers. 8
Figure 2.12. ATLAS's Electromagnetic and Hadronic Calorimeters. 9
Figure 3.1. Schematic of a wedge-shaped module of the TileCal Calorimeter. At the rear of the module there is a support girder that houses a unit called drawer. 10
Figure 3.2. TileCal's modules and the drawer units. The WLS fibres are connected to the PMTs through a circular opening in the drawer units that contain the photomultipliers and all the front-end electronics. 11
Figure 3.3. Segmentation in depth and η of the TileCal's barrel module (left) and extended barrel (right). 12
Figure 3.4. Glued fibres in the girder insertion tube. 12
Figure 3.5. Picture of the HV Micro board. (1) is the Motorola microcontroller, (2) are 256 KB flash memories, (3) are the 256 KB RAM, (4) is the 2 KB EEPROM, (5) are opto-couplers of the CANBus interface and (6) is the connector to the CANBus cable. 13
Figure 3.6. Picture of part of the HV Opto. (1) is one of the regulation loops, (2) is a opto-coupler, (3) are two 100 MΩ HV resistor, (4) are 6 DACs, (5) are two -5 V regulators, (6) is the ADC, (7) is a voltage reference and (8) is a 2 KB EEPROM. 14
Figure 3.7. Electric scheme of the regulation Loop. 14
Figure 3.8. Two partial views of the HV Bus. In the top, the low and high voltage input connector (1) and the connectors of the PMTs' HV (2). In the bottom picture, the connectors that link the HV Opto to the HV Bus (3) and the connector that links the two HV Bus boards (4). 14
Figure 3.9. Graph showing the predicted (blue) and achieved (green) integrated luminosity along the year of 2017. Image taken from [9]. 15
Figure 3.10. Super-drawer concept. In yellow new electronics, in green: adaptor boards for compatibility with the legacy system and in blue legacy system electronics. 16
Figure 3.11. Part of the electrical scheme of the new regulation loop. 17
Figure 4.1. Control tree of the first version of the HV Remote system. 19
Figure 4.2. Analogue circuitry of the HV Remote boards. These circuits are designed for one channel each. 20

Figure 4.3. Current control hierarchy of the HV regulation system. The FPGAs are connected to the DCS via ethernet and communicate with the HV Remote and Power Supplies boards via SPI. 21
Figure 4.4. Simplified scheme of the HV Remote's digital control circuit. The FPGA converts the commands from the PC to digital signals, that are sent to the digital chips (Expanders, ADC or DACs) via SPI. 22
Figure 4.5. Example of SPI Master connected to 3 Slaves. Source: Wikipedia page about the SPI interface. 23
Figure 4.6. SPI clock modes. When CPOL = 0 the clock is idle at '0' and activates at '1' and when CPOL = 1, the clock is idle at '1' and activates at '0'. When CPHA = 0, bits are clocked at the leading edge, and when CPHA = 1, bits are clocked at the trailing edge of clock signal. 24
Figure 4.7. Pinout of the MCP23S17. 25
Figure 4.8. Format of the Control Byte (device Opcode). The A2, A1 and A0 bits are the address of the device. The first 4 bits are fixed. 26
Figure 4.9. Format of the control byte and the Address Byte that contains the address of the register to whom the (read or write) operation is intended 26
Figure 4.10. Pin Configuration of the DAC7568 29
Figure 4.11. Overall architecture of DAC7568. 30
Figure 4.12. Format of the 32-bit input shift register. 31
Figure 4.13. Temporal diagram exemplifying the SYNC operation. The first write sequence is invalid, because SYNC is brought high before the 32nd falling clock edge. The second sequence is sent with success because SYNC stays in a low state along all the bits transfer. 32
Figure 4.14. Write sequence for enabling the internal reference (Static Mode). Corresponds to the hexadecimal word 08000001h. 32
Figure 4.15. Write sequence for disabling the internal reference (static mode). Corresponds to the hexadecimal word 08000000h. 32
Figure 4.16. Write sequence for enabling the internal reference (flexible mode). Corresponds to the hexadecimal word 09080000h. 33
Figure 4.17. Write sequence for the internal reference to be always enabled (flexible mode). Corresponds to the hexadecimal word 090A0000h. 33
Figure 4.18. Write sequence for the internal reference to be always disabled (flexible mode). Corresponds to the hexadecimal word 090C0000h. 33
Figure 4.19. Write sequence to switch from flexible mode to static mode. Corresponds to the hexadecimal word 09000000h. 33
Figure 4.20. Pin configuration of the MAX1240. 34
Figure 4.21. Serial interface timing sequence. 35
Figure 4.22. Pin configuration on the left, and functional block diagram of the bus switch on the right. 35
Figure 4.23. Pin configuration of the left and functional block diagram on the right. 36
Figure 4.24. Inputs (SCLK, DIN, CS_CARD, CS_SPA, CS_SPB, CS_SPC and CS_SPD) and outputs (DOUT and EXP_OUT) of the HV Remote. In the centre the 8-bit bus switch is shown, connected to the FPGA on its left. The black filled triangles are the digital grounds of the HV Remote. 37
Figure 4.25. Port Expander D connections. On the bottom left corner is the voltage supply V3 that supplies the port expanders with 3.3 V. The address code of this expander is 011, with A2 connected to DGND and A1 and A0 connected to 3.3 V. Pins 19 and 20 are left unconnected. 38

Figure 4.26. Connections of DAC1. The capacitors in the reference pin (pin 7) and in the power supply are recommendations from the datasheets. Pin 12 is the ground of the DAC which is connected to the common digital ground DGND. 39
Figure 4.27. Connections of the ADC. AIN is the analogue input voltage that is selected by the multiplexers. After reading and converting the analogue voltage to a 12-bit digital word, the ADC sends it to the master device (FPGA) via the DOUT line that is linked to the SPI MISO line... 40
Figure 4.28. Connections of MUX1. Pins 2, 3 and 13 are left unconnected. VP12 and VN12 are the analogue voltage supplies ±12 V of the HV Remote. 41
Figure 5.1. Raspberry Pi b+ pinout. The name of the pins is shown as BCMi, where i corresponds to the numbering of the GPIOs of BCM2835. 43
Figure 5.2. Raspberry Pi SPI test code. Highlighted by the rectangles are the initialization of the SpiDev object (blue), the connection to the SPI controller (orange), the definition of the max clock speed (green), the SPI transaction (red) and the closing of the SPI connection to the controller (white). A delay of 0.1 s is created and is used after each print, in the while cycle. 46
Figure 5.3. MCP23S17 class constructor. The arguments, in blue, represent the SpiDev instance (spiDevice), the port expander address code (device_id), the Pi's pin number associated with the port expander's RESET pin (pin_reset), the Pi's pin number associated with the port expander's CS pin (chip_select) and an instance of the RPi.GPIO class (iGPIO). 47
Figure 5.4. Content of the MCP23S17 constructor (the method __init__(self, spiDevice=spidev.SpiDev(), device_id=0x00, pin_reset=-1, chip_select=-1, iGPIO = GPIO)). 48
Figure 5.5. This is the "__writeRegister" private method. The first line of code defines the name of the method and the arguments it must be given. The second line asserts if the flag "isInitialized" has the Boolean value "True". The SPI transaction is performed by the "__spi.xfer2" line. 49
Figure 5.6. Content of the "setDirection" public method. Takes in as argument a pin number of the MCP23S17 GPIOs and one of the values DIR_INPUT and DIR_OUTPUT. The acceptable numbers for the "pin" argument range from 1 to 8 (GPBs) and from 21 to 28 (GPAs). 50
Figure 5.7. MCP23S17 class methods. The ones that are preceded by the underscore character, "__", are private methods, and the others are public methods. 51
Figure 5.8. Content of the "writeGPIO" method. It takes in as argument a 16-bit value ("data"). 51
Figure 5.9. Content of the "digitalWrite" method. Takes in as arguments a pin number of the MCP23S17 GPIOs, and one of the class' constants LEVEL_HIGH or LEVEL_LOW that correspond to '1' and '0' respectively. 52
Figure 5.10. DAC7568 class constructor. Each instance must receive as arguments a SpiDev instance ("spiDevice"), a MCP23S17 instance ("MCP") and a MCP23S17 pin number (SYNC). 53
Figure 5.11. Content of the DAC7568 constructor. It assigns the instance's arguments to attributes and creates additional attributes that act as flags to the remainder of the program or indicators of the state of the DAC's registers (DAC channel's register, internal reference's register and power-down logic's registers). 54
Figure 5.12. DAC7568 class methods. The most important are the "writeAndUpdateChannel" and "enableReference". 55
Figure 5.13. Content of the "__sendWord" private method. This method is responsible for performing all the DAC7568 SPI operations. It receives as arguments the different groups of bits of the DAC7568 32-bit input shift register. All the arguments must be 4-bit words, except for the "data" argument that must be a 16-bit word, where the first 12 bits are the valid sequence and the 4 last bits are ignored. 55

Figure 5.14. Code of the "writeAndUpdateChannel()" method. This method sets the respective bit groups to the correct values in order to perform a write and update channel operation. The method takes in as argument a letter (between A and H) and a value of 12 bits. On line 173 the argument "data" is transformed to a 16-bit value and stored in the variable "data_bits" which is the one that is inserted as argument in the "__sendWord" method. 56
Figure 5.15. MAX1240 class constructor. Arguments are a SpiDev instance ("spiDevice"), a pin number for the SHDN channel ("SHDN"), a pin number for the ADC chip-select ("chip_select") and a MCP23S17 instance ("MCP"). 57
Figure 5.16. Content of the "readVoltage" method. It returns the value read by the ADC in bits. 57
Figure 5.17. HVREMOTE class constants and methods. 58
Figure 5.18. HVREMOTE class constructor. It has a private attribute for each of the main components of the HV Remote boards (MCP, DAC and ADC). 59
Figure 5.19. SpiDev, GPIO and HVREMOTE initializations. 60
Figure 5.20. Set HV tab of the GUI. Now all channels are disabled, since all checkboxes are unchecked. The orange rectangle signals one checkbox, the blue rectangle signal the labels, the red signals the combo box that allows to select one of the 48 channels, the black rectangle highlights an entry that allows to insert a HV value and the green rectangles highlight the buttons. 61
Figure 5.21. The read HV tab. This will be where the real-time plots will be added. For now, only a channel selector (red), a button (green), two labels (blue) and a display (purple) were added to test the HVREMOTE class. 62
Figure 5.22. Code of the "Channel" class. The "change_state" method handles the event of checking/unchecking the checkboxes. The methods "check_box" and "uncheck_box" are used to change the boxes states internally. 62
Figure 5.23. Content of the "MainWindow" class. The constructor's code is presented. The class' methods definitions are also shown. 63
Figure 5.24. HV Remote's digital control emulated in a breadboard connected to the Raspberry Pi. The cables coming from the Pi (highlighted by the blue circle) are: the SPI MOSI (green), the SPI MISO (orange), the SPI SCLK (yellow), the chip-select of the port expanders (white), the 3.3 V (red and purple) and ground (black). Signalled by the red circle are the port expanders, signalled with the blue arrow is the DAC, signalled by the pink arrow is the ADC and signalled the green is the MUX. 64
Figure 5.25. Block Diagram of the breadboard setup. The orange line is the MISO connection, the yellow is the SPI clock, the green is the MOSI line, the blue is the port expander's shared chip-select, the purple represents the MUX's address lines, the red is the DAC's SYNC and the black is the DAC's output. 65
Figure 5.26. Testing the enable/disable channels. We can see that the LEDs are turned on, meaning that at that moment, those channels were enabled. 66
Figure 5.27. Set of write and read tests. It shows the value inserted by the user ("Channel i set to:") and the values read by the ADC ("Channel i reading:"). A difference of 20 to 30 counts can be seen between the value specified to the DAC and the one that is read by the ADC. 67
Figure 5.28. Write-Read test for all the DAC's eight channels. This test was made with a step of 15 counts and spans the entire valid range of data (from 0 to 4095). This test was made with a SPI clock frequency of 976 kHz. 68
Figure 5.29. Zoom of the middle region of the plot of Figure 6.28. 68
Figure 5.30. Plot of the error of the ADC readings versus value inserted to the DAC. This test was made with the same steps and same clock frequency as the ones above. 69

Figure 5.31. Plot of the error of the ADC readings versus value inserted to the DAC, with capacitors connected. This test was made with the same steps and same clock frequency as the ones above. 69
Figure 5.32. Error of the ADC's readings with capacitors connected and without the MUX. The DAC's channels were directly connected to the ADC. 71
Figure 5.33. SPI clock frequency tests. The plot of the right side is a zoom of the middle region of the left plot. These tests were made with a step of 1 count for the clock frequencies of 15.2, 61, 244 and 976 kHz. 71
Figure 5.34. SPI clock frequency tests. The plot of the left side shows the tests results for the clock frequencies of 1.953, 3.1, 7.8 and 15.6 MHz, and the plot of the right side shows the same results but without the 15.6 MHz data. 71
Figure 5.35. SPI clock frequency tests for 3.1 MHz and 1.953 MHz. This plot is a zoom of the middle region of the original plot, that shows the distribution of points with more detail. 72

Tables Index

Table 5.1. Digital coding to select the SPI mode 24
Table 5.2. Address mappings of all the registers, in the case where BANK = ‘1’ and BANK = ‘0’. 27
Table 5.3. IOCON register bits. The bit 0 is unimplemented. At POR the bits are ‘0’. 27
Table 5.4. IODIR register bits. At POR the bits are ‘1’ 28
Table 5.5. GPIO register bits. At POR the bits are ‘0’. 28
Table 5.6. Summary of all the registers with IOCON.BANK = 0 29
Table 6.1. SPI clock speeds. 45
Table 6.2. Summary of the mean error and mean error variation for a setup without capacitors and with capacitors. 70

Acronyms

ADC	Analog to Digital Converter
ALICE	A Large Ion Collider Experiment
ATLAS	A Toroidal LHC ApparatuS
CLK	ClocK
CMS	Compact Muon Spectrometer
CPHA	Clock PHAse
CPOL	Clock POLarity
CS	Chip Select
DAC	Digital to Analog Converter
DCS	Detector Control System
GPIO	General Purpose Input Output
GUI	Graphical User Interface
HL	High Luminosity
HL LHC	High Luminosity Large Hadron Collider
HV	High Voltage
HVLV	High Voltage and Low Voltage
LHC	Large Hadron Collider
LHCb	Large Hadron Collider beauty
LSB	Least Significant Bit
LV	Low Voltage
MISO	Master Input Slave Output
MOSI	Master Output Slave Input
MSB	Most Significant Bit
MUX	MUltipleXer
POR	Power On Reset
SCLK	Serial CLocK
SPI	Serial Peripheral Interface
SSH	Secure Shell

1 Introduction

1.1 Scope of this work

The ATLAS detector of the LHC in CERN, is under an upgrade program which includes improvements to its hadronic barrel calorimeter, named TileCal. This calorimeter has a cylindrical geometry with a tile structure. The tiles are made of a material that interacts with particles resultant from the collisions in the LHC, releasing high-frequency radiation. This radiation is collected by wave-length shifting fibers that convert the absorbed radiation to visible light and send it to photomultipliers tubes that convert it into an electrical signal. These photomultiplier tubes need a high voltage supply, which an HV regulation system provides.

The High-Luminosity LHC project predicts an increase in radiation, due to its aim of increasing the luminosity, which is an important factor for the performance of accelerators. This will have an impact on the HV regulation system's electronics that are already obsolete. So, it was proposed to develop a new radiation-hard system with newer components. Two solutions are on their way: one on-detector solution, based in the old system but with replaced components, and a remote system complemented by multi-conductor cables that distribute the HV to the PMTs.

The remote system constitutes 256 pairs of boards, each pair comprising one HV distribution board (HV Remote) and one power supply board (Power Supplies). The boards will be controlled by 16 FPGAs (each controlling 16 pairs of boards). The communication between the boards and the FPGAs will be made by an SPI interface and the communication with the ATLAS DCS (Detector Control System) is made via an ethernet protocol.

1.2 Objectives

This work focuses on developing a software to control the (HV Remote) boards of the remote HV regulation system, that is being developed with elements of LIP (Laboratory of Instrumentation and Experimental Particle Physics) and DF-FCUL (Physics Department of Faculty of Sciences of the Lisbon University), and on the development of a GUI (Graphical User Interface) that will be used to functionally test the boards before they are sent to CERN. Besides, the boards will also undergo some temperature tests in order to see how its components behave at different temperature regimes.

The development of the software interface required a good understanding of the digital control circuits of the HV Remote boards, which includes a good knowledge of how the communication with its components is established.

1.3 Software implementation

In this work, instead of an FPGA, a Raspberry Pi was chosen to be used, although in the future the software must be adapted to the FPGA. The Raspberry Pi tests the whole control hierarchy of the remote system, before using the FPGA. The Pi also offers a very versatile and friendly user set of GPIO ports, which allow us to control two or three boards at the same time and perform several tests to the software, while it is being developed. Because of this, the software will be implemented directly in the Pi, using version 3 of the Python language. The Pi's Python libraries come with two very useful modules that will be used in this project: the SpiDev module, that

establishes communication with the SPI peripherals of the Pi, therefore allowing us to transfer data with the HV Remote boards, without accessing the Pi's hardware, and a RPi.GPIO library (that will be imported as simply GPIO), that directly controls the state of the Pi's GPIO ports.

The GUI will be written in Python 3 as well, but using an additional module, specifically designed for graphical interface developments. This module is named PyQt5, which implements a C++ library, Qt5, in the Python language. This module offers several functionalities that will be useful to build a good test interface. It is based on a set of layout and widget objects, where the widgets can be buttons, checkboxes, lists, insert lines, and other kind of user interactive objects, and the layouts allow us to freely place the widgets in any part of our application's window.

1.4 State of the HV Remote board

When this thesis was completed, one HV Remote board has arrived, which will be the target of several tests. The digital control will only be tested after the most important connections have been verified and after guaranteeing the good functioning of the power supplies. A picture of this board is shown below. The software that was developed in this thesis will then be tested in this board, to test the operations on all the 48 channels.

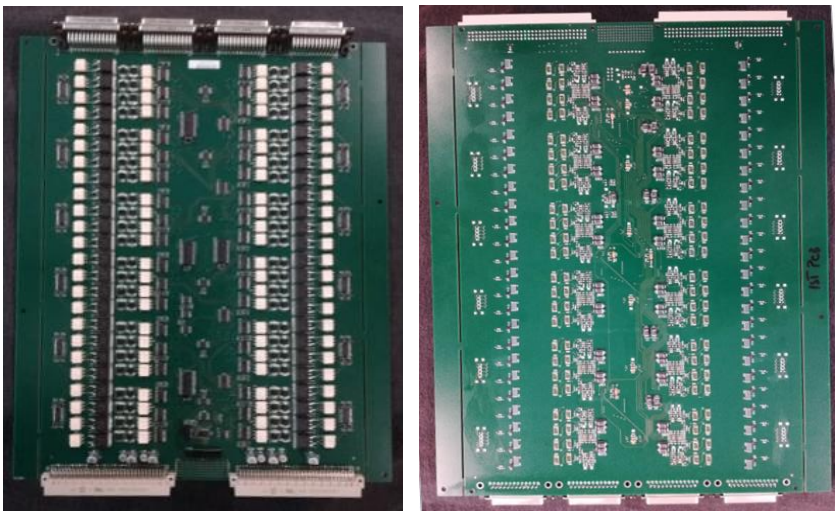

Figure 1.1. Frontal picture of the HV Remote board on the left, and of the backward part on the right.

1.5 Document structure

This document consists of 6 chapters, including this introduction. In the second chapter a description of the ATLAS and its sub-detectors is made. The third chapter focus on the TileCal calorimeter, giving a brief explanation of its mechanical structure, the tiles, the wavelength shifting fibers and the old HV regulation system of the PMTs. Chapter 4 mentions the high-luminosity upgrade that is being made to the LHC and its impact on the HV regulation system, pointing the key points that must be improved. Chapter 5 gives a description of the HV Remote board, and of its digital components. Finally, chapter 6 offers a brief explanation of the Raspberry Pi and explains how the software interface and the GUI were developed. In the end some tests to the GUI's features are presented. This work ends with a conclusion in chapter 7, that summarizes what was accomplished and predicts future work.

2 The Atlas Experiment

2.1 Overview of the ATLAS detector

The LHC (Large Hadron Collider) in CERN, is the largest particle accelerator in the world. It is a circular accelerator with a length of 27 km and is located 175 m deep beneath the surface. Its purpose is to accelerate beams of particles, thus increasing their energies, and collide them in the regions of its detectors.

There are four locations where the collision may happen, which correspond to 4 different detectors: ATLAS, CMS, ALICE and LHCb. ATLAS is a general-purpose detector of the LHC, that investigates a wide range of physics, from the search for the Higgs boson to extra dimensions and particles that could constitute the dark matter [1]. It is the largest volume particle detector ever built, until this date. The collisions that occur at the centre of ATLAS can reach energies of the order of TeV for p-p (proton – proton) collisions and hundreds of TeV for A-A (heavy ion – heavy ion) collisions. These will result in the creation of numerous particles that will fly in all directions. Some of the properties of these particles, such as momentum and energy will be measured by six different subsystems that are arranged in layers around the collision point.

The conditions present in the LHC's experiments and the phenomena resultant from the particle collisions imposes some requirements on ATLAS [2] [3]:

- It requires fast, radiation-hard electronics and sensor elements.
- High detector granularity, which help to handle particle fluxes and to reduce the influence of overlapping events.
- Large pseudorapidity[1] coverage for all azimuthal angles. The azimuthal angle is measured around the beam axis.
- Good charged-particle momentum resolution.
- Good electromagnetic calorimetry, which is necessary for electron and photon identification.
- Full-coverage hadronic calorimetry for accurate jet and missing transverse energy measurements.
- High-precision muon momentum measurements, with the capability to guarantee accurate measurements at the highest luminosity (highest particle flux).
- Highly efficient triggering, since there will be a great quantity of events, so the detector must be able to efficiently select the ones with the most interest for the experiment.

A view of the overall layout of ATLAS is shown in Figure 2.1. It is composed of several cylindrical sections surrounding the interaction point (the location where the particles collide).

[1] Pseudorapidity, η, is a measurement that is related to the angle that particles, resultant from the collisions, make with respect to the beam axis. It is defined as $\eta = -\ln(\tan \theta/2)$, where θ is the polar angle from the beam axis.

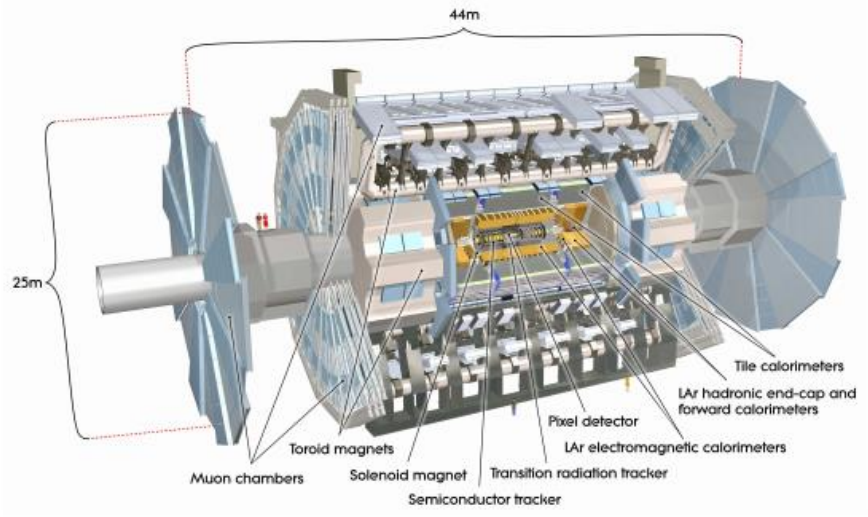

Figure 2.1. The ATLAS Detector layout.

The detector is composed of 4 main sub-detectors:

- The inner detector.
- The electromagnetic and hadronic calorimeters.
- The muon spectrometer.
- The magnetic system.

The magnetic system comprises a thin superconducting solenoid that surrounds the inner detector cavity and three large superconducting air-core toroids encompassing the calorimeters.

The inner detector is contained in a cylindrical cavity, located at the central part of the detector. It is composed of a discrete combination of high-resolution semiconductor pixel and strip detectors in its inner part and straw-tube tracking detectors in its outer part.

The electromagnetic calorimetry is composed of three (one barrel and two end-caps) high granularity liquid-argon (LAr) sampling calorimeters. The hadronic calorimetry is composed of a barrel calorimeter with a scintillator-tile structure and two LAr hadronic calorimeters at the end-caps. There are also two LAr forward calorimeters that provide both electromagnetic and hadronic measurements and extend the pseudorapidity coverage.

The calorimeters are surrounded by the muon spectrometer, that defines the overall dimensions of ATLAS, that presents a chamber structure, each with its own purpose. The muon spectrometer uses the magnetic fields generated by the magnetic system to perform muon momentum measurements.

2.2 The Magnet System

The ATLAS superconducting magnet system's main task is to create two different magnetic fields that will allow the identification of particles and the measurement of its momentum. It features four large superconducting magnets (see Figure 2.2): one solenoid and three air-core toroids.

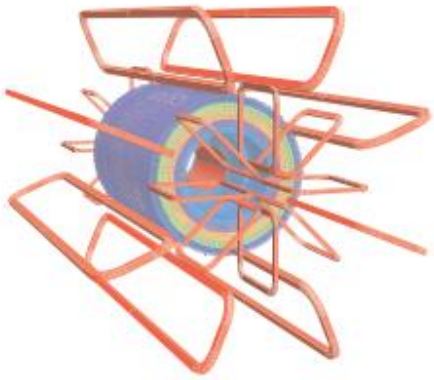

Figure 2.2. Geometry of the magnetic system. In the centre is the solenoid system and around it, is the barrel toroid. At the sides are the end-cap toroids.

Figure 2.3. Image of the central Solenoid in the factory.

Surrounding the inner detector and aligned with the beam axis is the central solenoid, Figure 2.3, that was positioned in between the two end-cap electromagnetic calorimeters. This demanded a minimisation of the material thickness, in order to achieve the desired calorimeter performance.

The system of three large air-core toroids (one barrel and two end-caps) surround the central solenoid and generate the magnetic field for the muon spectrometer. The two end-cap toroids (Figure 2.5) are inserted at each end of the barrel toroid (Figure 2.4), lining up with the central solenoid. Each of the three toroids consist of eight coils assembled radially and symmetrically around the beam axis.

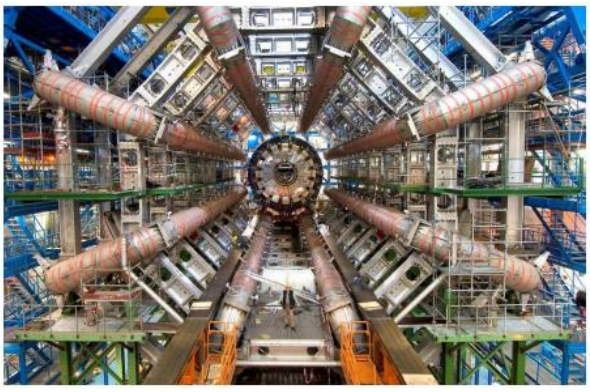

Figure 2.4. Picture of the Barrel Toroid.

Figure 2.5. Picture of one of the end-cap toroids.

2.3 The Inner Detector

The inner detector is placed within a cylindrical cavity in the innermost part of ATLAS. Surrounding it, is the magnetic field generated by the central solenoid (section 2.2). The inner detector's main function is to provide momentum and tracking measurements with high resolution. It also provides electron identification over a wide range of energies. The inner detector consists of three independent but complementary sub-detectors (Figure 2.6). In the inner radii a pixel detector and a semiconductor tracker (SCT) detector provide high-precision measurements, and in the outer radii a transition radiation tracker (TRT) arranged in straw tubes, provides continuous tracking measurements.

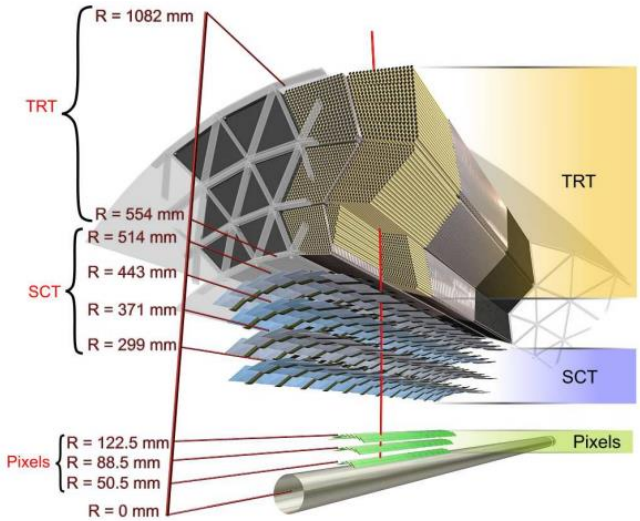

Figure 2.6. A representation of the sub-detectors of the inner detector and their radial positions inside the cylindrical cavity.

The mechanical layout of the inner detector is divided into three parts: one barrel extending over and two identical end-caps covering the rest of the cylindrical cavity. In the barrel region, the layers of the sub-detectors are arranged on concentric cylinders around the beam, while in the two end-caps the tracking detectors (SCT and TRT) are arranged in disk layers perpendicular to the beam axis, as shown in Figure 2.7.

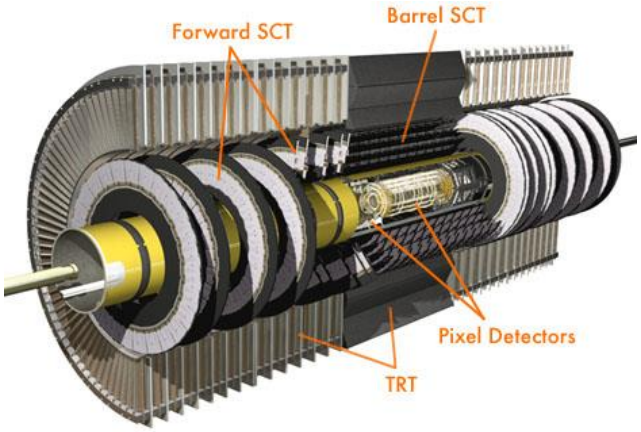

Figure 2.7. Mechanical structure of the inner detector.

2.3.1 The Pixel Detector

The pixel detector, **Figure 2.8**, is designed to provide a set of very high-granularity and high-precision measurements close to the interaction point. Because of this, it will be exposed to great quantities of radiation, so it requires the use of radiation-hard materials for its construction. The whole sub-detector was placed in the central barrel of the inner detector. There, pixel layers segmented in $R\phi$ and z coordinates, are arranged in 3 concentric cylinders around the beam, as shown in Figure 2.6 and **Figure 2.8** (barrel layers 0, 1 and 2). At the end-caps of this central barrel, 5 pixel-layers in a disk format are placed perpendicular to the beam axis, on each side.

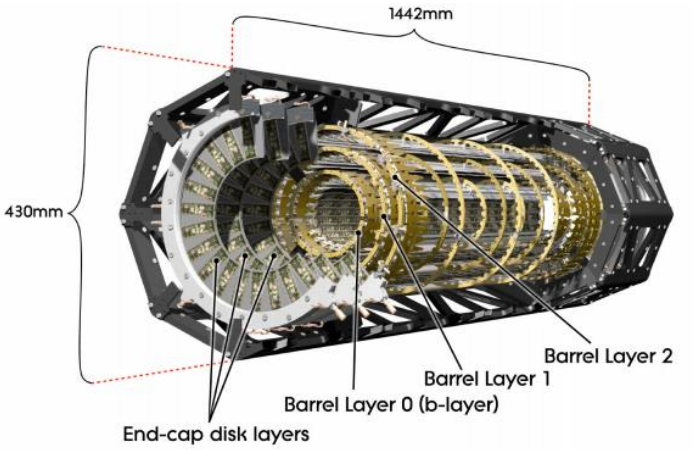

Figure 2.8. Scheme of the pixel detector. In the centre are the 3 pixel-layers, concentric to the beam axis and in the extremities are the 5 end-cap disk layers.

2.3.2 The semiconductor tracker (SCT)

The SCT is designed to provide precision tracking measurements in the intermediate radial length of the inner detector, and momentum, impact parameter, vertex position and pattern recognition measurements. A section of it is placed in the central barrel and the others are at the end-cap regions of the inner detector.

The barrel SCT consists of eight layers of silicon microstrip detectors that provide precision points in the $R\phi$ and z coordinates, using small angle stereo strips. The layers are mounted in 4 carbon-fibre cylinders, concentric to the beam axis. The end-cap modules use tapered strips, with one set aligned radially, and are mounted up in three rings onto nine disks, as shown in Figure 2.9.

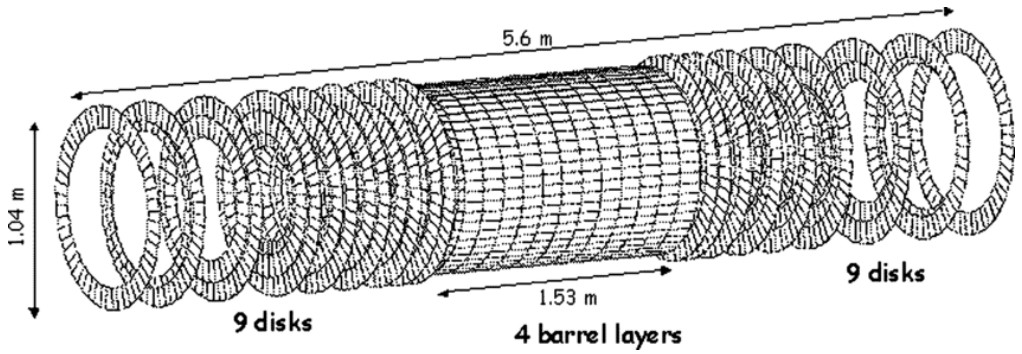

Figure 2.9. Layout of the semiconductor tracker (SCT).

2.3.3 The Transition Radiation Tracker (TRT)

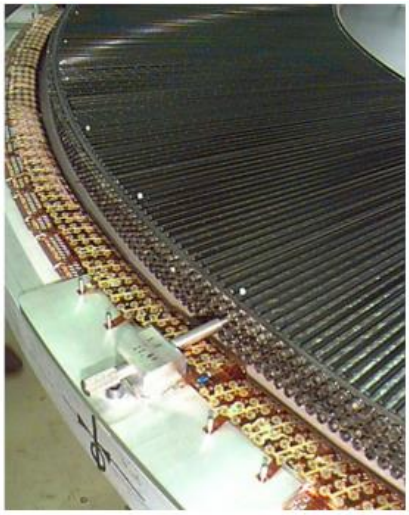

Figure 2.10. Straw tubes mounted in one of the end-cap TRT wheels.

The TRT consists on a series of straw tube detectors, each having 4 mm in diameter, designed to give information in the $R\phi$ coordinate. It also contributes significantly to momentum measurements and its straw tubes also contain xenon gas, that allows for electron identification.

In the barrel region, the straws are aligned parallel to the beam axis, while in the end-cap regions the straws are arranged radially in wheels (part of one is shown in Figure 2.10).

2.4 The Muon Spectrometer

The muon spectrometer forms the outer part of ATLAS and it is designed to detect charged particles exiting the barrel and end-cap calorimeters and to measure their momentum. It is based on the magnetic deflection of muon tracks in the superconducting air-core toroid magnets of the magnetic system of ATLAS (chapter 2.2).

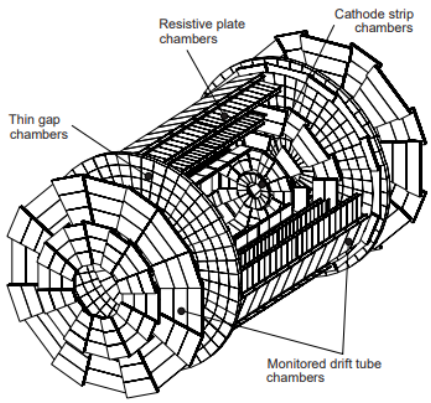

Figure 2.11. Layout of the muon chambers.

Precision measurements of the muon tracks are provided by two main chambers: the monitored drifted tubes (MDTs), which perform measurements in the principal bending direction of the magnetic field, and the cathode strip chambers (CSCs) that have a higher granularity and are used at larger pseudorapidity and closer to the interaction point. A trigger system composed of two main chambers, the resistive plate chambers (RPCs) that are used in the barrel, and the thin gap chambers (TGPs) that are used in the end-cap sections, provide track information, thus complementing the precision measurement chambers In the barrel the chambers are arranged in concentric cylinders with the beam axis. At the end-cap regions they are arranged in four disks also concentric to the beam axis. A layout of the muon spectrometer's chambers is shown in **Figure 2.11**.

2.5 Calorimeters

The calorimetry of ATLAS, **Figure 2.12**, consists of two main sampling calorimeter systems: the electromagnetic that measure the energy of electrons and photons and the hadronic calorimeters that measure the energy of particles that interact via the strong force (hadrons). These are composed of smaller sections of sampling detectors with full φ-symmetry. The calorimeters closest to the beam axis are housed in three cryostats: one barrel that contains the electromagnetic barrel calorimeter, and two end-caps, each containing an electromagnetic end-cap calorimeter (EMEC), a hadronic end-cap calorimeter (HEC) located behind the EMEC and a forward calorimeter (FCal), placed in the region closest to the beam. These calorimeters use liquid argon (Lar) as the active medium. Finally, in the outer region there is a hadronic sampling calorimeter with a cylindrical geometry that uses a structure of plastic scintillator tiles as the active medium and steel as the absorber. This calorimeter is designated TileCal and it will be described in more detail in chapter 3.

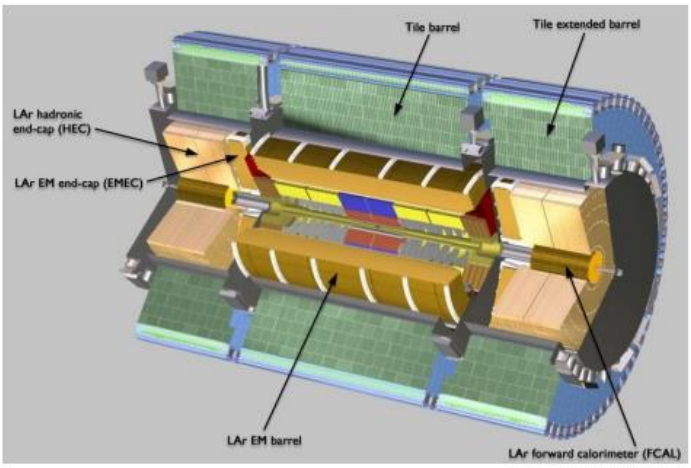

Figure 2.12. ATLAS's Electromagnetic and Hadronic Calorimeters.

3 The TileCal Calorimeter

3.1 Overview of the calorimeter

The TileCal [4][5] is a hadronic sampling calorimeter. Its main task is to provide precision measurements of hadrons and missing transverse energy and energy reconstruction of the jets produced in p-p interactions. Being a hadronic calorimeter means that it is designed to measure particles that interact via the strong nuclear force. The TileCal forms the inner shell of the ATLAS detector, encompassing all ATLAS sub-detectors, except for the muon spectrometer.

The TileCal uses a laminate of steel plates of various dimensions as the absorber material and a scintillator tile structure as the active medium. The highly periodic structure of the system allows the construction of the detector in a modularized way. The scintillating tiles are placed in a plane perpendicular to the beam and are radially staggered in depth. The scintillating tiles are readout by wavelength shifting (WLS) that deliver the light to photomultipliers (PMTs) which amplify and convert the optical signal into an electrical signal. Each scintillating cell is readout by 2 PMTs.

3.2 The Mechanical Structure of TileCal

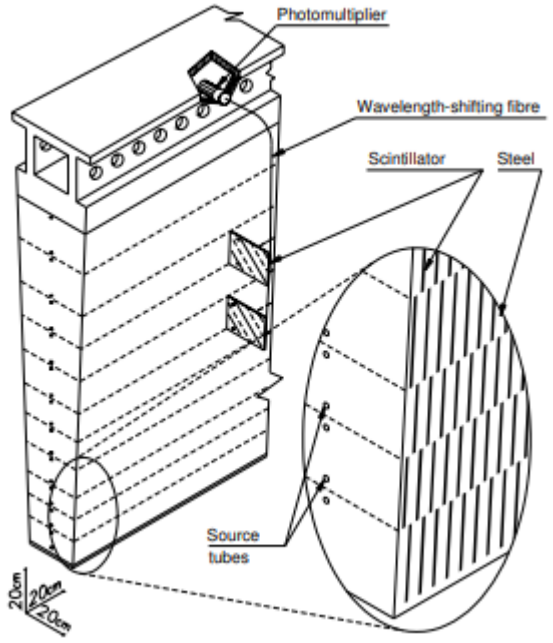

Figure 3.1. Schematic of a wedge-shaped module of the TileCal Calorimeter. At the rear of the module there is a support girder that houses a unit called drawer.

The TileCal has a cylindric geometry concentric to the beam axis and is subdivided into a central barrel and two extended barrels. Each barrel consists of 64 wedge-shaped modules, shown in Figure 3.1, each covering the azimuthal angle, of $2\pi/64 \approx 0.1$. The assembled modules form a periodic structure of alternating layers of iron plates and scintillating tiles, allowing for a good sampling frequency and a compact calorimetry.

At the rear of each module there is a support girder, where some units called drawers are inserted. These units are mechanically divided in two but work together as a single unit (the whole unit is usually called super-drawer). They contain the photomultiplier tubes, the wavelength shifting fibers and all the front-end electronics, which includes the readout electronics and the high voltage regulation system of the PMTs. The low-voltage power supplies which power the readout electronics are mounted in an external steel box, which has the cross-section of the support girder.

The TileCal's structure is self-supporting through a bearing connection between modules at the inner radius and a bolted plate connection at the outer radius, where the girder is mounted.

The calorimeter is assembled by mounting and bolting the modules to each other, and the modules are built by bolting the girders to the steel-scintillating structures. In the end, shims are inserted in the inner and outer radius load-bearing surfaces to control the overall geometry and yield a nominal module-module azimuthal gap of 1.5 mm. Figure 3.2 shows the mechanical sections of TileCal, its wedge-shaped modules and the drawer units.

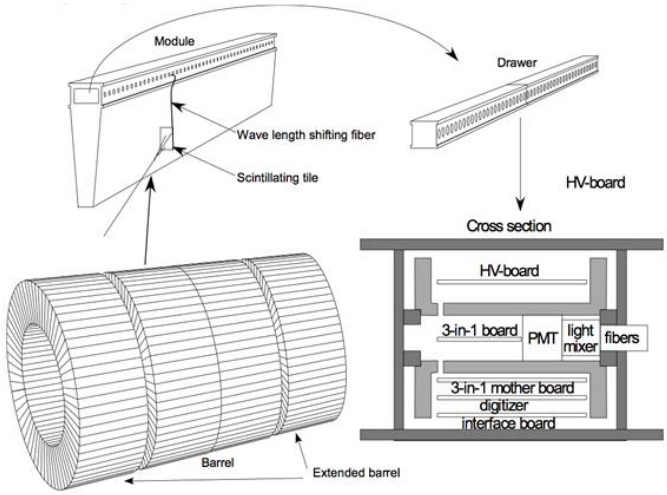

Figure 3.2. TileCal's modules and the drawer units. The WLS fibres are connected to the PMTs through a circular opening in the drawer units that contain the photomultipliers and all the front-end electronics.

3.3 Optical Elements: the scintillating tiles, the wavelength shifting fibres and the photomultipliers.

The active medium of the TileCal calorimeter consists of several plastic scintillating tiles with eleven different trapezoidal sizes (one for each depth in radius) [6]. The scintillating tiles are radially staggered in depth and normal to the beam line and are made of polystyrene (a polymer with a light emission peak around the 420 nm). Particles crossing the tiles will induce the production of ultraviolet light, that will be collected and subsequently converted to visible light by the wavelength shifting fibres. Light is collected at the tiles' edges and propagates along the fibres by total reflection, being transmitted to a PMT.

The TileCal's wedge-shaped modules are segmented in depth and in η, thus forming three different layers (A, BC, D) with different interaction lengths (Figure 3.3). This results in a cell structure, where each cell has a transversal segmentation. The cells are obtained by grouping a specific bundle of fibres, that bring light from one side of a group of tiles, to the same PMT. Each cell is readout by two PMTs, with each PMT reading one of its two sides.

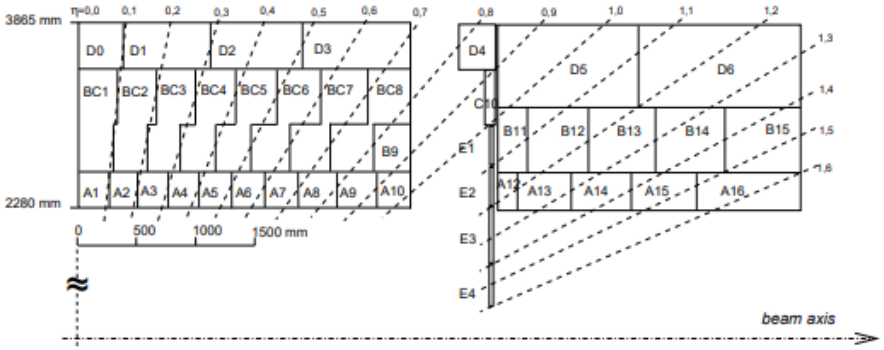

Figure 3.3. Segmentation in depth and η of the TileCal's barrel module (left) and extended barrel (right).

After the insertion of tiles and fibres into a module, the bundles of fibres are glued together into a fibre-insertion tube (Figure 3.4). These tubes are then fixed into the girder holes as shown in Figure 3.1, where they will serve as the optical interface to the PMTs.

The PMTs and the front-end electronics are housed in the mechanical units named super-drawers, which are inserted in the rigid steel girder. The drawers can be extracted of the girder, in order to perform maintenance on the PMTs, or on the electronics, during shut-down periods. The high-voltage regulation system, needed to supply and regulate the high voltage of the PMTs' anodes, is also included in the front-end electronics.

Each super-drawer is mechanically divided into two smaller drawers that contain 24 PMTs each, thus each super-drawer contains a maximum of 48 PMTs. One drawer is needed in order to perform a reading of an entire module of the extended barrels, while for reading one module of the long barrel, two drawers are needed. In total, the TileCal has approximately 10000 PMT channels.

Figure 3.4. Glued fibres in the girder insertion tube.

3.4 The old HV distribution system of the PMTs: HV Opto and HV Micro boards

The HV distribution system [7] is responsible for supplying, monitoring and setting the high voltage of approximately 10^4 PMTs. The voltage can be changed to any value in the interval from -500 to -900 V. In order to have stable measurements, the high voltage supplied to the PMTs must also be very stable, because a small change in the HV will induce a large gain variation.

The hardware of this system is located inside the super-drawer units that are divided into an internal and an external drawer. These drawers can house at maximum 24 PMTs. In total there are 256 drawers that are inserted on the girder of the 64 wedge-shaped modules.

The specifications of the HV system are:

- To individually adjust the HV of the TileCal's PMTs;
- The applied HV must be stable with a spread smaller than 0.5 V and a ripple smaller than 20 mV peak to peak, in order to ensure the stability of the PMT's voltage.
- Sensitivity to temperature variations smaller than 0.2 V/ºC and insensitivity to humidity values up to 60%;
- Insensitive to magnetic field values up to 0.1 T;
- Able to deliver an electric current of 350 μA to each PMT;
- Able to switch off some PMT channels, in order to reduce the impact on cell energy measurements.

The system was designed to 48 PMTs and had one HV power supply, that can supply the PMTs with either -830 V or -950 V, per super-drawer. The old HV regulation system consisted on: two HV Opto boards, one for each drawer; one HV Micro board; two long HV bus boards, one in each drawer; one short bus, HV Flex, that links the internal and external HVBus boards; and seven temperature probes, integrated on all HV boards that are monitored by the HV Micro.

The HV Micro board, shown in Figure 3.5, is the one that processes all the information. It is responsible for commanding the HV Opto boards, monitoring the input HV and LV (low voltage) of the power supplies, converting the DAC and ADC's (located in the HV Opto) words to physical values and to make the link of the entire system to the DCS. The communication with the ATLAS DCS is established using a CANBus network [8] that is connected to a CANBus interface, integrated in the HVMicro's microcontroller, through an opto-isolated interface.

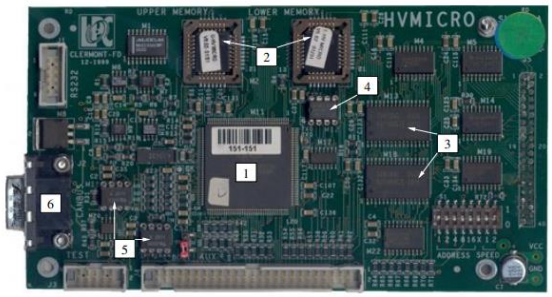

Figure 3.5. Picture of the HV Micro board. (1) is the Motorola microcontroller, (2) are 256 KB flash memories, (3) are the 256 KB RAM, (4) is the 2 KB EEPROM, (5) are opto-couplers of the CANBus interface and (6) is the connector to the CANBus cable.

The HV Opto boards, Figure 3.6, are responsible for regulating the HV of 24 PMTs. To each channel is associated a regulation loop and a 12-bit DAC, as shown in Figure 3.7, that allow this the voltage regulation. The 24 PMTs are divided into two groups of 12, which are connected to emergency switches that enable/disable the groups' channels. Besides adjusting the HV and enabling/disabling the PMTs, the HV Opto boards also perform voltage readings to the PMTs' channels and temperature probes.

Each HV Opto board contains only one ADC. To perform the necessary operations on the 24 channels, 3 multiplexers are used. The HV Opto also has a reference voltage to monitor the stability of the ADC.

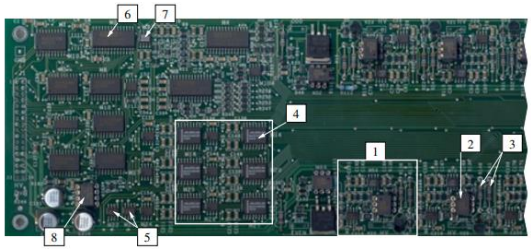

Figure 3.6. Picture of part of the HV Opto. (1) is one of the regulation loops, (2) is a opto-coupler, (3) are two 100 MΩ HV resistor, (4) are 6 DACs, (5) are two -5 V regulators, (6) is the ADC, (7) is a voltage reference and (8) is a 2 KB EEPROM.

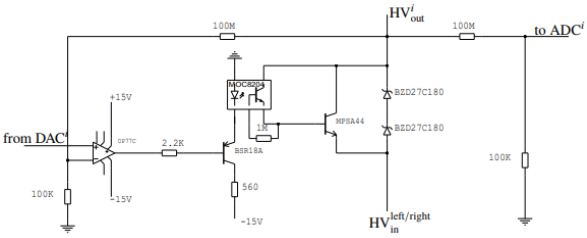

Figure 3.7. Electric scheme of the regulation Loop.

The HV Bus boards, Figure 3.8, purpose is to provide mechanical support for the HV Micro and HV Opto boards and to make the connections between these two boards and between the HV Opto and the PMTs. Each HV Bus board also contains one temperature probe.

All the HV systems of the TileCal's partitions are controlled and monitored independently by the DCS that uses a WinCC OA application. So, the user uses the DCS to perform the required operations, like reading the voltage of a PMT channel or adjusting its voltage and enabling/disabling channels. The DCS sends this command to the respective HV Micro that interprets the information and commands the HV Opto boards to perform the specified operations.

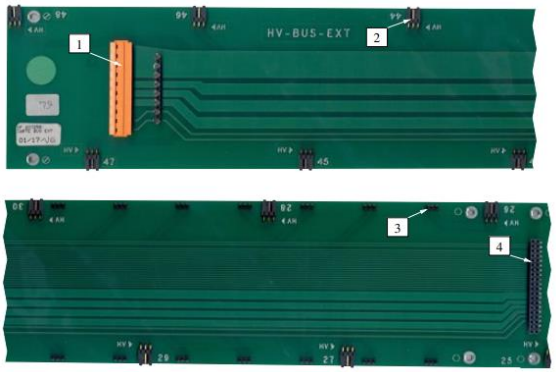

Figure 3.8. Two partial views of the HV Bus. In the top, the low and high voltage input connector (1) and the connectors of the PMTs' HV (2). In the bottom picture, the connectors that link the HV Opto to the HV Bus (3) and the connector that links the two HV Bus boards (4).

3.5 The High-Luminosity Upgrade of the LHC

The LHC luminosity exceeded its target in 2017, producing up to 60 collisions at each bunch crossing, with each bunch containing around 100 billion particles [9]. It has provided its two main experiments (ATLAS and CMS) with approximately 50 fb^{-1} (unit used to measure integrated luminosity) of data, which corresponds to 5 billion million collisions. Figure 3.9 shows a graph where the predicted and achieved integrated luminosity for the year of 2017 are compared. As we can see the performance of the LHC far exceeded the predicted one.

Figure 3.9. Graph showing the predicted (blue) and achieved (green) integrated luminosity along the year of 2017. Image taken from [9].

Since the LHC has had an excellent performance, providing its experiments (ATLAS for example) with a large amount of data, including the discovery of the Higgs boson, CERN scientists are convinced that they can push its performance even further. This upgrade of the LHC will result in an increase of luminosity (which is the main indicator of the performance of an accelerator) and the energies involved in the collisions. This will allow physicists to study mechanisms and physical theories in much greater detail, such as the recently discovered Higgs boson, as well as to discover new rare phenomena that might reveal itself in future experiments.

The materials budget for the upgrade is set at 950 million Swiss francs between 2015 and 2026. The HL-LHC (High Luminosity LHC) should be operational around 2026 [10].

3.5.1 The ATLAS upgrade for the HL-LHC

ATLAS (chapter 2) was initially designed to study proton-proton collisions at the LHC with center of mass energies up to 14 TeV at a maximum luminosity of 10^{34} $cm^{-2}s^{-1}$. The LHC is undergoing a series of upgrades with the intent of increasing its luminosity by 10-fold. This will be a great challenge for ATLAS, since it will require significant detector optimizations, changes and improvements, specially to the systems located at the inner radii, due to the increase of radiation resultant from collisions. Consequently the Inner Detector, the forward calorimeter and forward muon wheels will be the most affected systems, by the increasing radiation and particle fluxes, while the barrel calorimeters and muon chambers are expected to handle the conditions of the HL-LHC, so they will not be modified (although they will need upgrades).

The ATLAS upgrade is planned in three phases: phase 0, phase 1 and phase 2 [11]. In phase 0 a new cooling system for the Inner Detector and a new neutron shielding for the Muon Spectrometer will be implemented. On phase 1 the end-cap module of the muon spectrometer, will be replaced by a new one. On the new MSW will be installed new detector technologies, like the Micromegas chambers and the Small Strip Thin Gap Chambers [12]. Phase 2 will mainly see Inner Detector and calorimeter upgrades, as well as improvements on the trigger and data acquisition system. At the end of this phase the integrated luminosity should reach its target value for the HL-LHC. The upgrades of the ATLAS calorimeters will mainly consist on an upgrade or replacement of the readout electronics of all calorimeters and a replacement of the cold electronics inside the LAr Hadronic end-cap calorimeters. The upgrades to the TileCal will be described with more detail in the next section.

3.5.2 The upgrades to the Tile Calorimeter

One of the objectives of the upgrades to the TileCal is to have an easier maneuverability of the drawer units. To accomplish this, the super-drawer will comprise four independent mini-drawers that will host up to 12 PMTs (Figure 3.10). This change will improve the reliability of the cooling circuits and will grant an easier access to the front-end electronics. Other changes will consist on an upgrade of TileCal's on- and off-detector electronics, due to its obsolescence and a need for an increase in reliability and radiation tolerance.

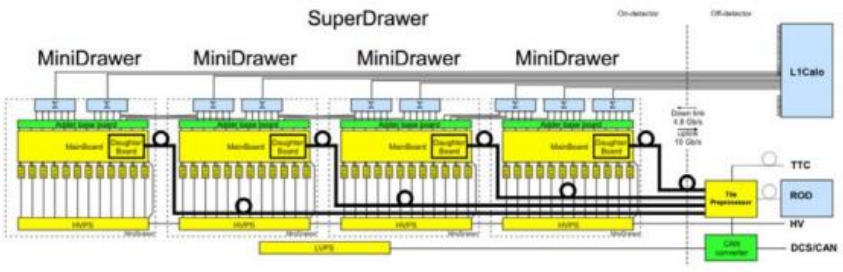

Figure 3.10. Super-drawer concept. In yellow new electronics, in green: adaptor boards for compatibility with the legacy system and in blue legacy system electronics.

It is proposed that each mini-drawer will contain (besides the 12 PMTs): 3in1 front-end boards to shape and amplify the PMTs signals, one mainboard that will control, monitor and readout the front-end boards; one daughterboard that will receive digitized signals from the mainboard and

will perform the communication between the front-end and off-detector electronics; and one low voltage power supply that supplies the on-detector electronics with +10 V. To evaluate this concept a demonstrator is being designed with a compatibility with the legacy system, in order to be tested in the current TileCal [14].

The high voltage power supply system has two possible solutions for its upgrade. One consists on an on-detector solution that uses one HV board per mini-drawer to provide high voltage control up to 12 channels. These boards must be radiation hard and must communicate with the off-detector side via the daughterboard. The other solution, which is the scope of this work, is based on a remote system, where the regulation boards would be placed away from the detector, and the high voltage would be distributed to the PMTs via long multi-conductor cables.

There will also be a change to the voltage dividers of PMTs. Whereas before there was passive dividers that shared the voltages between the photocathode, the 8 dynodes and the anodes of the PMTs, some will be replaced by active dividers (transistors and diodes). The usage of these transistors will increase the high voltage noise, and as such, both solutions must change the regulation loop of the HV boards. After some tests [15], it was concluded that the old electrical scheme of the regulation loop, Figure 3.7, should be changed to the one of Figure 3.11, where the transistor near the opto-coupler was removed.

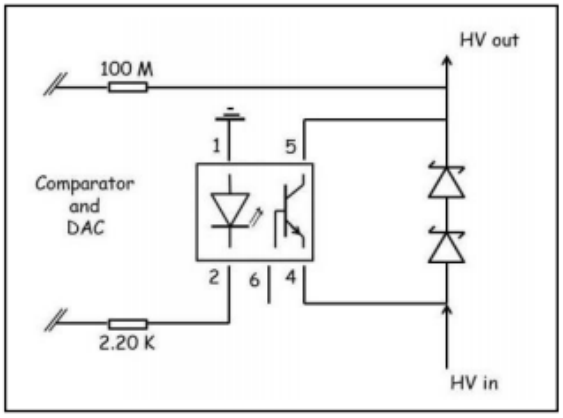

Figure 3.11. Part of the electrical scheme of the new regulation loop.

The remote system project will take a different approach for the communication interface with the DCS. The main advantage of this solution is that the electronics won't be vulnerable to radiation, which will increase its performance and durability. Instead of using a dedicated controller board, as was done in the old system with the HV Micro, this solution proposes the use of several pairs of boards, each comprising a HV distribution and regulation board (HV Remote) and a power supply board (Power Supplies), that provides the necessary low voltage for the HV Remote's components and a fixed high voltage value (either -830 V or -950 V). These boards will be controlled by FPGA development boards via the SPI communication protocol and will communicate with the ATLAS DCS through an ethernet network.

4 The HV Remote system

Answering the demands of the High-Luminosity upgrade of ATLAS, a new system for the high voltage control of the TileCal's PMTs was proposed. It consists of several HV Remote boards that regulate and monitor the high voltage of the TileCal's PMTs, each associated with a power supplies board (Power Supplies), that provides a fixed HV value (-830 V or -950 V) and the LV necessary for the HV Remote. These boards are going to be controlled by FPGA development boards that will be connected to the DCS via an ethernet network. All the boards will be allocated in crates (each containing up to 16 HV Remote and their dedicated power supplies boards, and one FPGA Development board) in the ATLAS's US15, a room at 100 m from the detector, where the amount of radiation does not become a critical problem for the electronics. This also has the advantage of having a permanent access to the room, in order to perform an easier maintenance to the system, during the LHC shutdowns. The regulated high voltage will be distributed along multi-conductor cables of approximately 100 m length. This new HV remote system will also have a different communication interface with the DCS, an on/off control for each PMT's channel and the small change in the regulation loop that was described in section 3.5.2.

This chapter will start with an explanation of the control hierarchy, followed by an explanation of the interface that is going to be used to communicate with the boards. After this, the main digital components used in this project will be explained. Finally, the electric schematics of the HV Remote boards and the purpose of each digital component will be explained in the end of this chapter.

4.1 General Description of the HV Remote System

Each HV Remote board will be able to regulate the HV of 48 PMTs (unlike the old HV Opto boards that only regulated 24 PMTs each). In total, 256 boards will be needed for the control of all the TileCal's 9852 photomultipliers. They will be stored in crates where each crate will contain 16 HV Remote boards. Unlike the HV Opto (section 3.4), the HV Remote will be able to individually enable/disable the PMTs' channels. Although the HV Opto also had the enable/disable feature, it could only be used on the odd or even channels of each 24 PMTs. This improvement of the HV Remote boards will result in a lower consumption and will prevent the acquisition of incoherent data.

Initially the control hierarchy of the new system was as shown in Figure 4.1. The PC workstation, configured as a node of the DCS of ATLAS, would receive the commands from the DCS via ethernet, and would resend them to a tree of ethernet switch ports, that would select the board to which the command was intended to. Attached to the HV Remote boards was a Tibbo module [16], that would receive the commands from the PC Workstation, interpret them and perform the necessary operations on the digital components of the board.

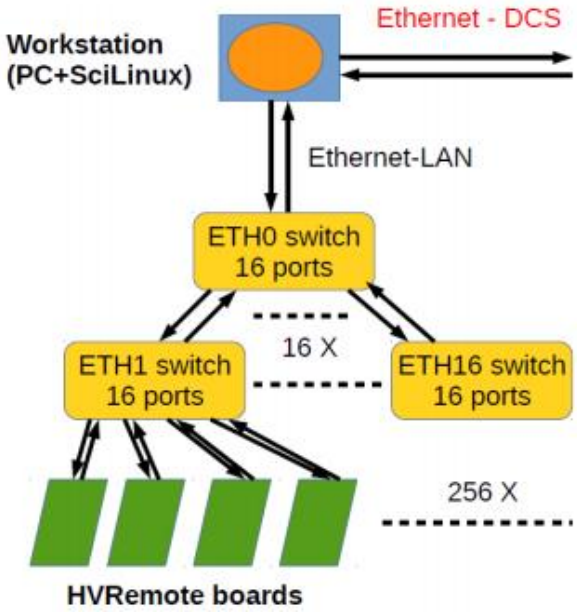

Figure 4.1. Control tree of the first version of the HV Remote system.

The communication between the Tibbo module and the digital components was established via an SPI (Serial Peripheral Interface) protocol, which is a synchronous serial communication, based on a master-slave architecture. The Tibbo module acted as the master and the HV Remote's chips acted as the slaves. The SPI will be explained in more detail in section 4.2.

In the initial design the HV Remote boards were still designed for 24 PMTs and the control circuit was composed of:

- Three 16-bit serial-parallel port expanders (MCP23S17 [17]) to implement the enable/disable feature and to control the other components.
- Three 12-bit DACs (DAC7568 [18]) to adjust the PMT's high voltages.
- Two temperature probes (TMP17 [23]) to measure the board's temperature.
- One test voltage of 1.2 V (AD589 [24]).
- One 12-bit ADC (TLV2541 [19]) and two 16-bit MUXs (MPC506 [25]) responsible of measuring the PMT's voltages, the temperature probes and the test voltage.
- One embedded Tibbo EM1206 module, that established the communication between the PC and the board.
- One DC/DC converter (MAX 3002 [26]). This was necessary to convert the digital signals because the digital signals of the HV Remote were of 5 V and the Tibbo signals were of 3.3 V.

These were all new components that were chosen in order to replace the obsolete ones that were present in the HV Opto boards and to change the digital control. It was expected that these will guarantee a long period of operation for the HV regulation system, without a need of replacement.

The analogue circuitry, which includes a read voltage loop, a regulation loop, and a switching channel loop, is designed for one channel. So, each HV Remote, initially had 48 of these analogue circuits, that are shown in Figure 4.2. There are three main signals: RD_CHi which are signals

proportional to the HV of the PMTs, that will be selected by the multiplexer and read by the ADC; the CHi_CTR, which comes from the output of the DACs and is used to regulate the HV of the PMTs; and the CHi_EN which are controlled by the GPIOs of the port expanders to enable/disable the PMTs.

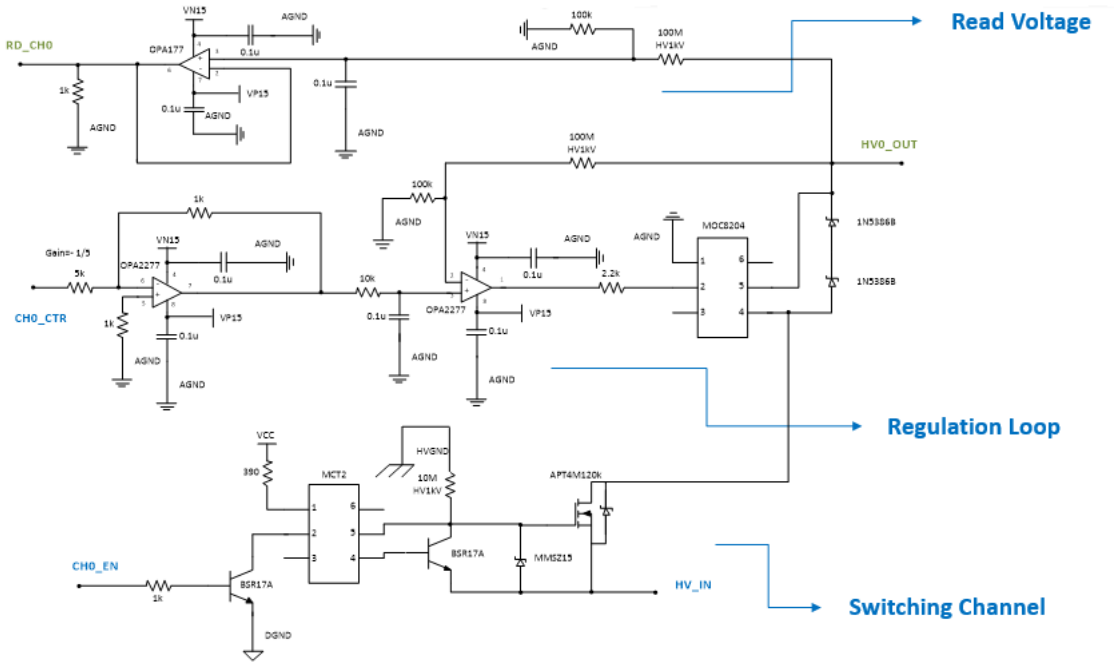

Figure 4.2. Analogue circuitry of the HV Remote boards. These circuits are designed for one channel each.

A prototype board (HV Remote-Ctrl [27]) with 24 channels was built to test the digital control and the analogue circuitry of the HV Remote, the Tibbo module and the ethernet and SPI protocols in order to evaluate if they were an adequate communication interface to this project. The results were satisfying since communications occurred with success and the board was regulating and reading properly the voltages of the channels. It was also concluded that the speeds achieved with the ethernet and SPI protocols were suitable to monitor in real-time the 256 HV Remote boards and all the TileCal's PMTs.

However, after some discussions, some changes to the overall system were implemented:

- First, the system on chip approach with the Tibbo module was replaced by 16 FPGA development boards, that will also be connected to the DCS via ethernet, where each FPGA development board will control 16 HV Remote as well as their respective Power Supplies boards. The communication between the FPGA and the HV Remote is still done via the SPI protocol. Overall the control hierarchy shown in Figure 4.1 was changed for the one of Figure 4.3. This change in the control hierarchy will have a more reliable communication and will allow easier maintenance to the system. The FPGA Bus boards contain the primary power supplies (12 V for the analogue components and 3.3 V for the digital ones), a power supply to the FPGA development board, two SPI buses (one to the HV Remote and another to the dedicated power supply board), 1 to 4 fanout buffers that create multiple copies of the SPI signals and distribute them among the HV Remote and the dedicated boards, and board selection and parallel expander buses.

- Second, the HV Remote boards were redesigned to regulate 48 PMTs, instead of 24. This was a bit of a challenge because of the limited available space in the boards. As a solution to this problem, the analogue components were replaced by quadruple versions, therefore reducing the number of components needed for the analogue circuitry.
- Third, and last, the digital components were picked in version with 3.3 V CMOS logic levels. This was an important decision, since eventually 5 V logic chips will disappear from the market (what would make the replacements hard), and because 3.3 V components have a lower consumption.

As was mentioned before, the boards are going to be stored in crates (16 HV Remote and dedicated power supplies for each crate), and the FPGA development and bus boards will be connected to the backplane of the crate.

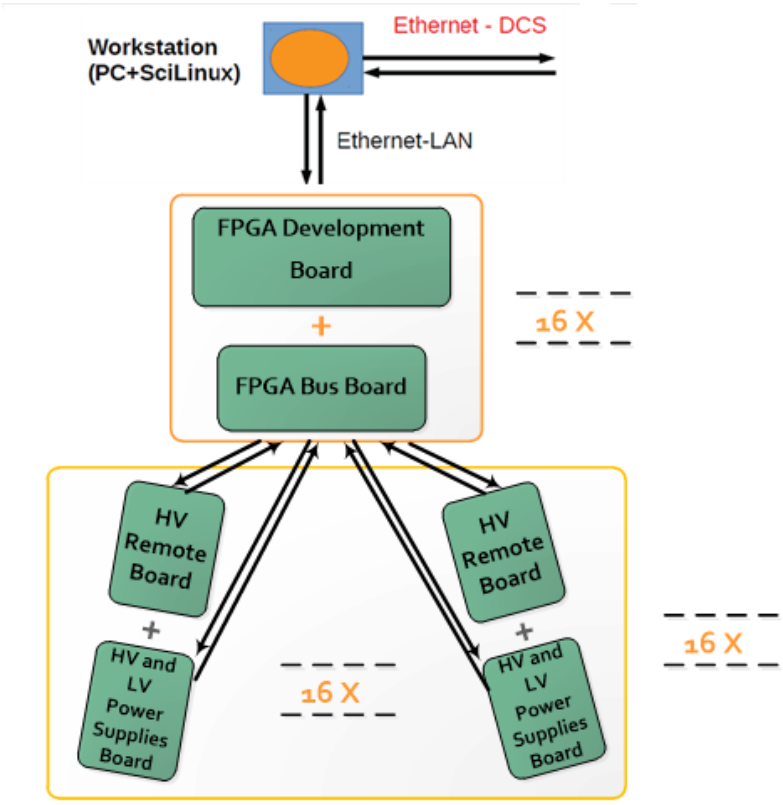

Figure 4.3. Current control hierarchy of the HV regulation system. The FPGAs are connected to the DCS via ethernet and communicate with the HV Remote and Power Supplies boards via SPI.

The integrated circuits used in the digital control of the new version of the HV Remote board are listed below. The port expanders were retained; regarding the DACs, although they are the same module, a new version was bought that offers a simplified operation. The multiplexers and the ADC were replaced by other components, and a bus switch was added to select to which board the system communicates. Each HV Remote board now has:

- 4 MCP23S17 (16-bit serial-parallel port expanders).
- 6 DAC7568 (12-bit octa-channel DACs).
- 1 MAX1240 (12-bit ADC).
- 4 MUX36S16 (16:1 analogue multiplexers).
- 1 SN74CB3Q3245 (8-bit digital bus switch).

The functions of these chips remain the same as the older version. A schematic of the control circuit is shown in Figure 4.4. The FPGA receives the commands from the PC workstation and converts them into digital words, that are sent to the board's digital chips via the SPI MOSI line. The digital bus switch is always the first component that the FPGA communicates with, since it is the one that selects to which board the communication is intended to. Then, the serial to parallel port expanders enables/disables the PMTs channels and selects which chips (ADC, DACs or MUXs) are to be active/inactive in order to receive/ignore the digital word sent from the FPGA. The DACs function is to adjust the PMTs voltage. The MUXs select one of the 48 PMT channels, which is to be read, and the ADC converts the analogue voltage of the selected channel to a digital word and subsequently sends it back to the FPGA via the SPI MISO line.

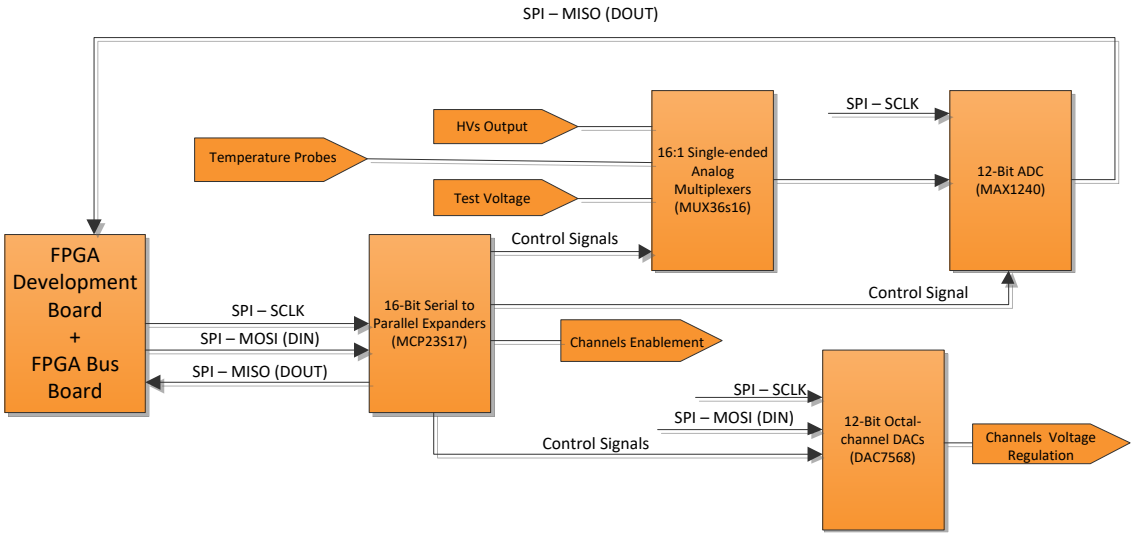

Figure 4.4. Simplified scheme of the HV Remote's digital control circuit. The FPGA converts the commands from the PC to digital signals, that are sent to the digital chips (Expanders, ADC or DACs) via SPI.

The temperature probes (TMP17) and the test voltage (AD589) used in the older version of the HV Remote were also retained. Their outputs are connected to the MUXs channels in order to be read by the ADC, when those channels are selected. The temperature sensors are going to be used in order to verify how much the board will heat in regions densely populated with chips. This information may be useful to optimize the cooling system of the crates in the future. The sensors can be used in environments with temperatures ranging from -40 ºC to 105 ºC.

4.2 The SPI protocol

The SPI (Serial Peripheral Interface) is a synchronous serial communication, that is composed of a master and one or several slaves. It is an interface with 4 lines: MOSI (Master Output Slave Input), MISO (Master Input Slave Output), SCLK (Serial Clock) and CS or SS (Chip Select or Slave Select). Communications always start by lowering the SS line. Then the master device controls the transfer of information. This means that the master is the one that generates and controls the clock signal, which is the one that controls the transference process of bits in and out of the master in the MISO and MOSI lines, respectively. The SPI was chosen due to its decent speed transmission, high throughput (the rate at which a successful message is delivered in a channel), simple hardware and software implementation and bidirectional data transmission (data can be sent and read at the same clock impulse). Although there are protocols with higher information transfer speeds, SPI's potential speed was enough for this project.

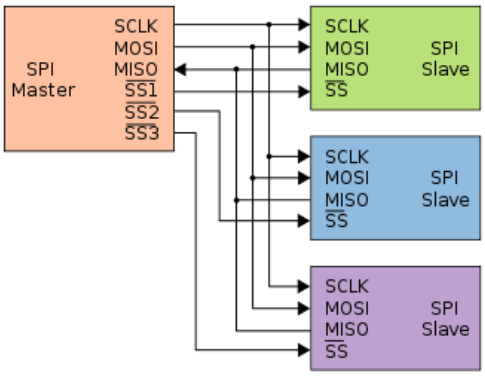

Figure 4.5. Example of SPI Master connected to 3 Slaves. Source: Wikipedia page about the SPI interface.

Another advantage of the SPI is that the MOSI, MISO and SCLK wires can be shared between the slaves, because each of them will have a unique SS wire (Figure 4.5) that allows to select only on device to communicate with. Therefore, every digital control device in the HV Remote boards will share the MISO, MOSI and SCLK lines. And all the HV Remote boards will also share the same SPI lines between each other. This is possible because each HV Remote will have a dedicated chip select signal. The CS/SS signals of the HV Remote boards will be controlled directly by the FPGA, and the CS/SS signals of the HV Remote's chips will be controlled by the port expanders, as was mentioned before in section 4.1.

The SPI clock signal has 4 different modes. Each device receives data in a specific mode, so care must be taken when communicating to each device, or else they won't work properly. These modes differ in the clock's phase (CPHA) and in the clock's polarity (CPOL) and can be selected by writing a digital word of two bits (or using an existent function in the master device), where the most significant bit corresponds to the value of CPOL and the least significant bit corresponds to the value of CPHA (Table 4.1). When CPOL is 0, it means the SCLK will have a positive polarity, when it transitions to the active state. If CPOL is 1, the SCLK will have a negative polarity when it transitions to the active state. When CPHA is 0, bits are transferred at the leading edge of the SCLK signals, and when CPHA is 1, bits are transferred at the tailing edge. A temporal diagram of these modes is show in Figure 4.6.

Table 4.1. Digital coding to select the SPI mode

CPOL	CPHA	Digital Word
0	0	00
0	1	01
1	0	10
1	1	11

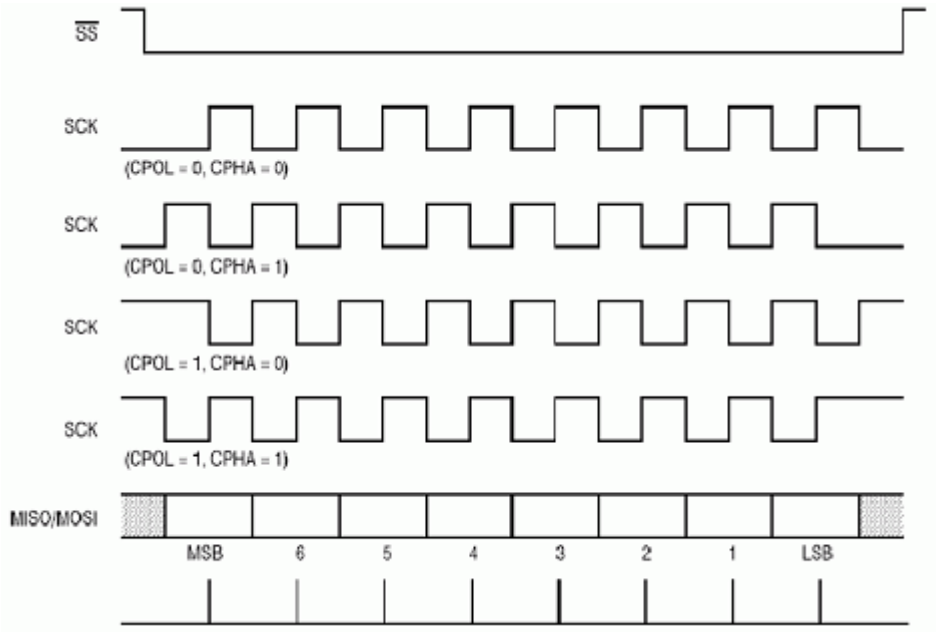

Figure 4.6. SPI clock modes. When CPOL = 0 the clock is idle at '0' and activates at '1' and when CPOL = 1, the clock is idle at '1' and activates at '0'. When CPHA = 0, bits are clocked at the leading edge, and when CPHA = 1, bits are clocked at the trailing edge of clock signal.

Besides the clock modes, the frequency of the clock signal must also be taken into consideration. On one hand we want to have the maximum frequency possible, because the HV Remote will need to perform lots of operations:

- Set the voltages in the DAC's channels
- Read the voltages of the PMTs' channels from the ADC, which requires also changing the address code of the MUXs
- Enable/Disable the PMTs' channels
- Read the board's temperature and the test voltage.

On the other hand, we must not surpass the maximum clock frequency for each of the devices, or else they wouldn't receive correctly the digital information. This requires a good knowledge of these chips, that are going to be explained in the next sub-chapter.

4.3 The Digital Devices of the HV Remote

4.3.1 The port expander MCP23S17

General Description

The MCP23S17, [17], is a 16-bit serial-parallel expander, whose main function is to convert the serial MOSI signal of the SPI protocol, to a parallel signal of 16 bits that is represented by the state of the GPIO (General Purpose Input/Output) ports of the MCP23S17. These ports can be configured either as inputs or outputs and are divided into two 8-bit port banks: PORTA and PORTB. Associated to each of these banks is an interrupt pin (INTA and INTB) that is not going to be used in this digital control system, so these pins are left unconnected. For the HV Remote project, these ports will be configured as outputs and will be used to enable/disable the PMTs channels, by setting their outputs to '1'/'0', and to control the chip select signals of the other HV Remote's chips.

The MCP23S17's SPI interface runs at a maximum clock frequency of 10 MHz and the proper SPI mode is 00 (CPOL = 0 and CPHA = 0). The SPI MISO, MOSI, SCLK and SS (or CS) signals are to be connected to the pins 14, 13, 12 and 11, respectively (pins SO, SI, SCK and $\overline{CS}$ of Figure 4.7).

The pins 9 (VDD) and 10 (VSS), correspond to the power supply and to digital ground of the circuit. In the HV Remote this chip will be supplied by 3.3 V.

The RESET pin is used to reset all the chip's registers to their default values. During operation, this pin must always be connected to the 3.3 V. When it is disconnected (or set into a low state), the MCP23S17 will reset all its registers, changing their current values to the register's default values.

This device also features a 3-bit address code (pins 15 to 17), that allows 8 expanders to be connected to the same chip select of the SPI. For now, a different chip select for each port expander is going to be used, but the possibility of using this address feature was not completely discarded yet. More tests will have to be made, to see if there is any advantage or disadvantage of using this method to address the port expanders.

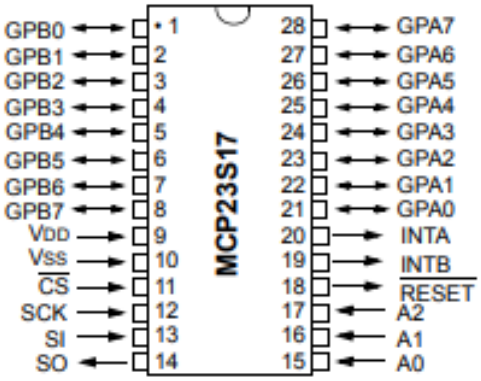

Figure 4.7. Pinout of the MCP23S17.

SPI Interface

The communication with the MCP23S17 is started by lowering $\overline{CS}$, and by sending three valid bit sequences (each of 8 bits). The first byte (word of 8 bits) clocked to the expander is called control byte (Figure 4.8). The control byte contains the address word of the device (that must match the externally biased pins A0, A1 and A2 of Figure 4.7) and the read/write bit, that will determine if it's a read or write operation. The address functionality may be enabled/disabled via the HAEN bit of the IOCON register (which is the register that configures the device). Also, even if the address feature is disabled the address pins must always be externally biased.

The second byte sent is the address of the register as shown in Figure 4.9. The addresses will depend on how the IOCON register was configured. The third byte contains the values that are intended to be written in the selected register. If it's a read operation the third byte isn't important, since the intention is to receive an information byte from the device. The three bytes may be sent without lifting the $\overline{CS}$ line.

The mode of the SPI must be 0 (equivalent to the digital word 00), which means that CPOL (Clock Polarity) and CPHA (Clock Phase) must both be 0. In this mode the clock starts at state '0' and performs a bit transfer when it transitions to '1'. As was already said, the maximum clock frequency for the MCP23S17 is 10 MHz. If information is clocked at a rate superior to this limit, the device will not communicate properly.

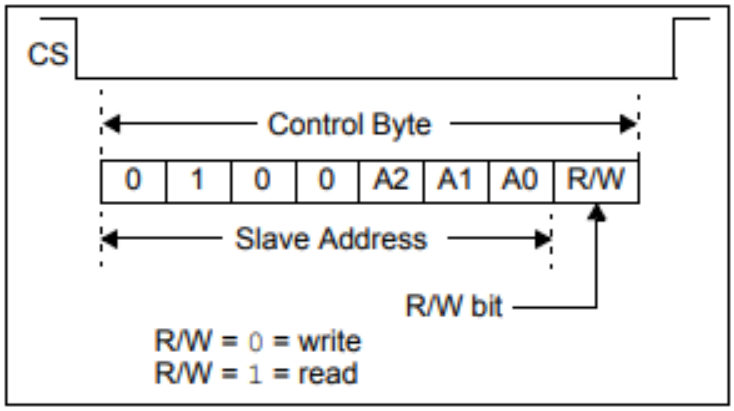

Figure 4.8. Format of the Control Byte (device Opcode). The A2, A1 and A0 bits are the address of the device. The first 4 bits are fixed.

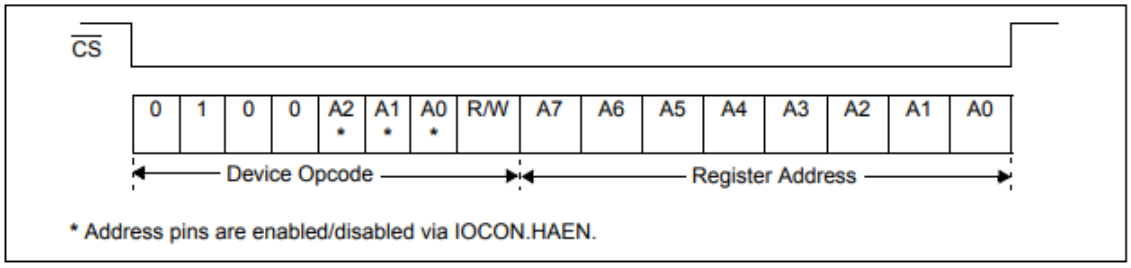

Figure 4.9. Format of the control byte and the Address Byte that contains the address of the register to whom the (read or write) operation is intended

The MCP23S17's registers

Table 4.2. Address mappings of all the registers, in the case where BANK = '1' and BANK = '0'.

Address IOCON.BANK = 1	Address IOCON.BANK = 0	Access to:
00h	00h	IODIRA
10h	01h	IODIRB
01h	02h	IPOLA
11h	03h	IPOLB
02h	04h	GPINTENA
12h	05h	GPINTENB
03h	06h	DEFVALA
13h	07h	DEFVALB
04h	08h	INTCONA
14h	09h	INTCONB
05h	0Ah	IOCON
15h	0Bh	IOCON
06h	0Ch	GPPUA
16h	0Dh	GPPUB
07h	0Eh	INTFA
17h	0Fh	INTFB
08h	10h	INTCAPA
18h	11h	INTCAPB
09h	12h	GPIOA
19h	13h	GPIOB
0Ah	14h	OLATA
1Ah	15h	OLATB

The MCP23S17 has 22 individual 8-bit registers (11 register pairs) with different functionalities as shown in Table 4.2. The registers are accessible by writing their addresses to the communication interface. Care must be taken when accessing the registers because there are two different address mappings. The mapping is chosen by the BANK bit of the IOCON register. There are also 2 operation modes: byte mode, where continuous operations can be performed on the same register and the sequential mode, in which after the transfer of each byte, the register address is incremented, so that the next byte will be transferred to a different register. In the later operational mode, after accessing the last register, the address pointer will roll over to the address 00h. In this project we will only use the byte mode.

Some important registers will be explained thereafter. Their default values at POR (Power On Reset) are shown in the next tables.

❖ IOCON – Input/Output Configuration Register

Table 4.3. IOCON register bits. The bit 0 is unimplemented. At POR the bits are '0'.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0
BANK	MIRROR	SEQOP	DISSLW	HAEN	ODR	INTPOL	—
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

IOCON is the configuration register. Each of its bits configures different functions of the device. The bits are writable and readable. The bit 0 is unimplemented so it doesn't matter.

- BANK (bit 7) – Controls how the registers are addressed. If the bit is '1', the A registers shall be numbered sequentially from '00h' to '0Ah' and the B registers from '10h' to '1Ah'. If the bit is '0' the registers shall be sequentially ordered in pairs (e.g.: IODIRA -> 00h, IODIRB -> 01h, IPOLA -> 02h, IPOLB -> 03h, …).
- MIRROR (bit 6) – Controls how the interrupt pins (INTA and INTB) relate with each other. If the bit is '1', the interrupt pins are functionally OR'ed (if INTA is 1, INTB will be 1 and vice-versa). If the bit is '0', they are independent of each other.

- SEQOP (bit 5) – Configures the mode of operation as Sequential or as Byte Mode. If the bit is '1' the device will operate in Byte Mode. If the bit is '0' the device will operate in Sequential Mode.
- DISSLW (bit 4) – Controls the slew rate function of the SDA/SI pin (which is the Serial Input). If the bit is '1' the slew rate will be controlled when driving from a high to a low state.
- HAEN (bit 3) – Enables/Disables the hardware addressing. The address pins (A2, A1, A0) must be externally biased, regardless of the HAEN bit value. If the bit is '1' the device's hardware address will match the pins value (A2A1A0). If the bit is '0' the device's hardware address will be '000'.
- ODR (bit 2) – Enables/Disables the INT pin as an open-drain output. If the bit is '1' the INT pin will be in open-drain output (overrides the INTPOL bit). If the bit is 0, the INT pin will be configured as Active driver output (INTPOL bit will set the polarity).
- INTPOL (bit 1) – This bit sets the polarity of the INT output pin. '1' is for active-high and '0' is for active-low.
- UNIMPLEMENTED (bit 0) – Read as '0'

❖ IODIR – I/O Direction Register

Table 4.4. IODIR register bits. At POR the bits are '1'

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

This register configures the pins as output or input. Each bit of this register corresponds to a GPIO pin. When the bit is '1' the pin is configured as an input, and when it's '0' the pin is configured as an output. The bits are writable and readable.

❖ GPIO – General Purpose I/O Port Register

Table 4.5. GPIO register bits. At POR the bits are '0'.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

This register reflects the value on the port. Reading from this register, we are reading the port. Writing to this register modifies the Output Latch (OLAT) register. The bits represent the logic levels ('1' = high and '0' = low).

Below is a table summarizing all the registers, their addresses with the BANK bit set as '0', and their POR values. We can see that each register has two Ports (A and B), except the IOCON register. Although Table 4.6 shows a IOCONA port and a IOCONB, those different addresses are actually pointing to the same register. The writing/reading procedures will be explained in full detail in chapter 5.

Table 4.6. Summary of all the registers with IOCON.BANK = 0

Register Name	Address (hex)	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	POR/RST value
IODIRA	00	IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0	1111 1111
IODIRB	01	IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0	1111 1111
IPOLA	02	IP7	IP6	IP5	IP4	IP3	IP2	IP1	IP0	0000 0000
IPOLB	03	IP7	IP6	IP5	IP4	IP3	IP2	IP1	IP0	0000 0000
GPINTENA	04	GPINT7	GPINT6	GPINT5	GPINT4	GPINT3	GPINT2	GPINT1	GPINT0	0000 0000
GPINTENB	05	GPINT7	GPINT6	GPINT5	GPINT4	GPINT3	GPINT2	GPINT1	GPINT0	0000 0000
DEFVALA	06	DEF7	DEF6	DEF5	DEF4	DEF3	DEF2	DEF1	DEF0	0000 0000
DEFVALB	07	DEF7	DEF6	DEF5	DEF4	DEF3	DEF2	DEF1	DEF0	0000 0000
INTCONA	08	IOC7	IOC6	IOC5	IOC4	IOC3	IOC2	IOC1	IOC0	0000 0000
INTCONB	09	IOC7	IOC6	IOC5	IOC4	IOC3	IOC2	IOC1	IOC0	0000 0000
IOCON	0A	BANK	MIRROR	SEQOP	DISSLW	HAEN	ODR	INTPOL	—	0000 0000
IOCON	0B	BANK	MIRROR	SEQOP	DISSLW	HAEN	ODR	INTPOL	—	0000 0000
GPPUA	0C	PU7	PU6	PU5	PU4	PU3	PU2	PU1	PU0	0000 0000
GPPUB	0D	PU7	PU6	PU5	PU4	PU3	PU2	PU1	PU0	0000 0000
INTFA	0E	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INTO	0000 0000
INTFB	0F	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INTO	0000 0000
INTCAPA	10	ICP7	ICP6	ICP5	ICP4	ICP3	ICP2	ICP1	ICP0	0000 0000
INTCAPB	11	ICP7	ICP6	ICP5	ICP4	ICP3	ICP2	ICP1	ICP0	0000 0000
GPIOA	12	GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0	0000 0000
GPIOB	13	GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0	0000 0000
OLATA	14	OL7	OL6	OL5	OL4	OL3	OL2	OL1	OL0	0000 0000
OLATB	15	OL7	OL6	OL5	OL4	OL3	OL2	OL1	OL0	0000 0000

4.3.2 The DAC7568

General description

The DAC7568, [18], is a 12-bit octal-channel DAC that features a 3-wire serial interface that is compatible with standard SPI (it contains wires for SCLK, for MOSI that is to be connected to D_{IN} and for the $\overline{CS}$ that is to be connected to $\overline{SYNC}$). The pin 2 (AV_{DD}) and pin 12 (GND) correspond to the voltage supply and digital ground, respectively. The power supply will be connected to the 3.3 V and GND to the common digital ground of the HV Remote board. Pins 3 to 6 and 8 to 11 are the eight outputs of this device, that will adjust the voltage of 8 PMT channels.

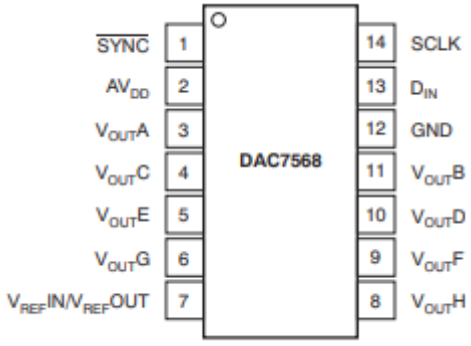

Figure 4.10. Pin Configuration of the DAC7568

This device includes a 2.5 V internal reference (pin $V_{REF}IN/V_{REF}OUT$) that is disabled by default. This internal reference has 2 different modes of operation that shall be explained later. If the user chooses to use an external reference, he can simply connect it to the $V_{REF}IN/V_{REF}OUT$ pin and maintain the internal reference disabled during operation. In the HV Remote the DACs will be using their internal references, so the modes of operation must be studied to decide which one shall be the most adequate for this project.

The DAC7568 also features different ways of updating the output voltage of its channels, and a way of powering them down in order to reduce current consumption. Additionally, this chip also contains a power-on-reset circuit that powers-up the DAC channels to either zero scale, midscale or full scale, until a valid write sequence is written.

The Figure 4.11. shows the architecture of the entire chip, as well as its pins. The Shift Register receives the bytes from the SPI line D_{IN} and configures the DAC Registers, or performs operations in the Power-Down Logic or in the Internal Reference. We can see that each channel has a data buffer where the information is stored before being sent to the DAC Register, that allow the user to write a value in the DAC's channels, even when they are powered down. When the channels are turned on again, their respective DAC registers will update their output voltage to the value that was stored in the Data Buffers.

The Figure 4.11 also shows the LDAC and CLR pins, but these are not present in the current DAC7568 model that is being used in this project, so they may be ignored.

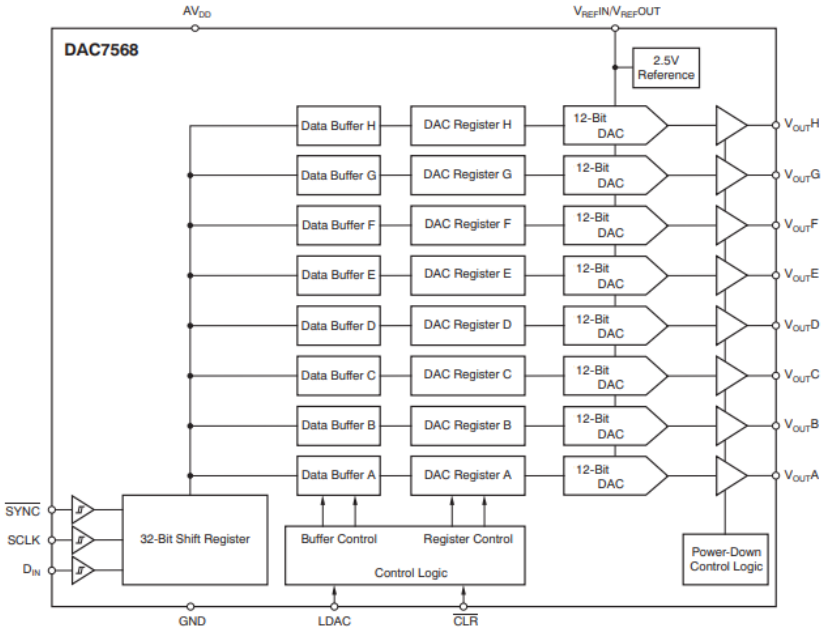

Figure 4.11. Overall architecture of DAC7568.

Serial Interface

As was stated before, the DAC7568 serial interface has 3 connections: SYNC, SCLK, and D_{IN}. All of them are inputs for enabling communication (or activating the chip), for the SPI master clock and for the data bytes sent via the MOSI line, respectively. They can also be used for other serial interfaces such as QSPI, Microwire Interface standards and DSPs[2]. The rate at which bits

[2] Digital Signal Processing

may be transferred on this interface can be as high as 50 MHz, which is higher than the one of the port expanders.

This chip's input shift register, shown in Figure 4.12, is 32-bits wide and can send data of 12 bit words (which ranges the integers interval from 0 to 4095). The conversion from these integer numbers (called DAC counts) to volts is given by equation (4.1), where V_{REF} is the voltage reference of the DAC, and Gain is a characteristic of the device, which in this case is 1.

$$V_{OUT} = \left[\frac{D_{IN}}{2^n}\right] \times V_{REF} \times \text{Gain} \qquad \textbf{(4.1)}$$

The 32 bits shift register consist of 4 prefix bits (Bit31 to Bit28), 4 control bits (Bit27 to Bit24), 4 address bits (Bit23 to Bit20), 16 data bits (Bit19 to Bit4) and 4 feature bits (Bit3 to Bit0). The 16 data bits comprise the 12-bit input code that ranges from Bit19 to Bit8. The bits from Bit7 to Bit4 are ignored by the device, when writing data to one of the channels, although they are used in other operations.

To write a sequence to the chip, first the SYNC line must be brought low. Data starts to transfer at each falling edge of SCLK, which means this device operates in SPI mode 0b01, where CPOL = 0 and CPHA = 1 (section 4.2). The sequence always begins by bit31, which is the Most Significant Bit, and in all operations, this bit must be set to '0'. The remaining prefix bits are ignored.

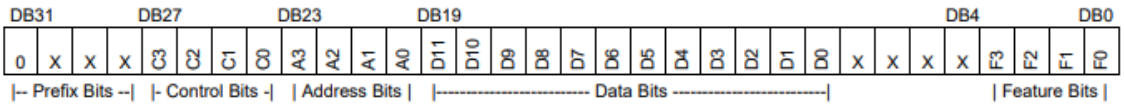

Figure 4.12. Format of the 32-bit input shift register.

When we want to write a certain value into one of the DAC7598 channels, the input shift register reads the control bits, after ignoring the last 3 prefix bits, in order to configure the DAC's operating mode. Then the address bits are read to select the DAC's channel. After this, the next 16 bits are read, and the Data bits are taken to update the selected channel's voltage. Finally, the last 4 bits (feature bits) are ignored. A new write sequence will start on the next falling edge of SYNC.

The SYNC line can also be used as an interrupt to stop communication with this device at any time. If the SYNC line is brought high before the 32nd falling clock edge, the SPI interface will be reset, and no data transfer will occur. In this sense SYNC may be interpreted as a hardware interrupt. An example of the SYNC line interrupting communications and of a successful communication is shown in Figure 4.13.

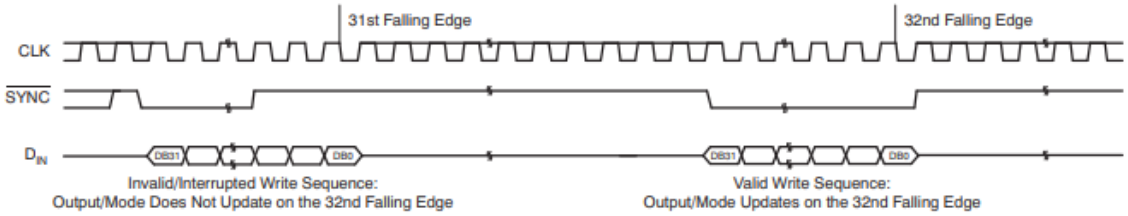

Figure 4.13. Temporal diagram exemplifying the SYNC operation. The first write sequence is invalid, because SYNC is brought high before the 32nd falling clock edge. The second sequence is sent with success because SYNC stays in a low state along all the bits transfer.

Internal Reference

The DAC7568 offers an internal reference of 2.5 V that is disabled by default. When it is disabled, an external reference may be connected to the $V_{REF}IN/V_{REF}OUT$ pin. If the internal reference is enabled, a switch is closed, thus connecting the internal reference to the $V_{REF}IN/V_{REF}OUT$ pin. The $V_{REF}IN/V_{REF}OUT$ pin must not be externally and internally connected at the same time!

There are two modes that allow communication with the internal reference: static mode and flexible mode. These modes determine how the internal reference will operate, when the DAC's channels are powered up/down. When turning on the device, the internal reference is disabled (default mode), until a valid write sequence to enable it is sent to the device.

In static mode, the internal reference is enabled by writing the 32-bit sequence in Figure 4.14. When all DACs power down, the reference is automatically powered down. When any of the DACs powers up the internal reference automatically powers up. To disable the internal reference, the 32-bit sequence in Figure 4.15, must be sent.

DB31			DB27					DB23				DB19														DB4				DB0
0	X	X	X	C3	C2	C1	C0	A3	A2	A1	A0	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	F3	F2	F1	F0
0	X	X	X	1	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
|-- Prefix Bits --|			|- Control Bits -|				| Address Bits |			|---------- Data Bits ----------|															| Feature Bits |			

Figure 4.14. Write sequence for enabling the internal reference (Static Mode). Corresponds to the hexadecimal word 08000001h.

DB31			DB27					DB23				DB19														DB4				DB0
0	X	X	X	C3	C2	C1	C0	A3	A2	A1	A0	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	F3	F2	F1	F0
0	X	X	X	1	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0
|-- Prefix Bits --|			|- Control Bits -|				| Address Bits |			|---------- Data Bits ----------|															| Feature Bits |			

Figure 4.15. Write sequence for disabling the internal reference (static mode). Corresponds to the hexadecimal word 08000000h.

The flexible mode has two methods of enabling the internal reference:

- Method 1 – The internal Reference is enabled by writing the 32-bit serial command shown in Figure 4.16. Like in the static mode, the internal reference will be disabled if all DAC channels power down and will automatically enable if any of the channels powers up.

- Method 2 – By writing the 32-bit serial command of Figure 4.17, the internal reference will always be enabled, regardless of the DAC's channels state (powered down/up). When performing a power cycle reset to the device (turning off and on the device) the internal reference will still be switched off.

The write sequence of Figure 4.18, disables the internal reference, and it will remain disabled (regardless of the DACs state) until a power cycle reset is performed to the device.

When the internal reference is operated in flexible mode, the static mode is disabled, so the sequence of Figure 4.19 allows to change from flexible mode to static mode. To change from static to flexible mode we only must write one of the enable/disable sequences of the flexible mode. The mode that is going to be used in this project is the one of Figure 4.17.

DB31			DB27				DB23				DB19													DB4				DB0
0	X	X	X	C3	C2	C1	C0	A3	A2	A1	A0	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	F3	F2	F1	F0
0	X	X	X	1	0	0	1	X	X	X	X	1	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
|-- Prefix Bits --|			|- Control Bits -|			| Address Bits |			|---------- Data Bits ----------|															| Feature Bits |			

Figure 4.16. Write sequence for enabling the internal reference (flexible mode). Corresponds to the hexadecimal word 09080000h.

DB31			DB27				DB23				DB19													DB4				DB0
0	X	X	X	C3	C2	C1	C0	A3	A2	A1	A0	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	F3	F2	F1	F0
0	X	X	X	1	0	0	1	X	X	X	X	1	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
|-- Prefix Bits --|			|- Control Bits -|			| Address Bits |			|---------- Data Bits ----------|															| Feature Bits |			

Figure 4.17. Write sequence for the internal reference to be always enabled (flexible mode). Corresponds to the hexadecimal word 090A0000h.

DB31			DB27				DB23				DB19													DB4				DB0
0	X	X	X	C3	C2	C1	C0	A3	A2	A1	A0	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	F3	F2	F1	F0
0	X	X	X	1	0	0	1	X	X	X	X	1	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
|-- Prefix Bits --|			|- Control Bits -|			| Address Bits |			|---------- Data Bits ----------|															| Feature Bits |			

Figure 4.18. Write sequence for the internal reference to be always disabled (flexible mode). Corresponds to the hexadecimal word 090C0000h.

DB31			DB27				DB23				DB19													DB4				DB0
0	X	X	X	C3	C2	C1	C0	A3	A2	A1	A0	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	F3	F2	F1	F0
0	X	X	X	1	0	0	1	X	X	X	X	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
|-- Prefix Bits --|			|- Control Bits -|			| Address Bits |			|---------- Data Bits ----------|															| Feature Bits |			

Figure 4.19. Write sequence to switch from flexible mode to static mode. Corresponds to the hexadecimal word 09000000h.

Write Commands

This device has different ways of setting the voltage of its channels. It has two functions:

- Write to an input register: this function allows us to write to the DAC register/buffer, where the voltage value is stored.
- Update register: this updates the voltage of the channel according to the digital value that is stored in its respective DAC register/buffer.

With these two functions it is possible to write to several registers without setting their channel's voltage. When we want to update the voltage, we only need to use the respective channel update write sequence. There is also a sequence that does these two operations (write and update) to all the DACs. The disadvantage of this is that the value will be the same to all channels.

The DAC7568 also offers some codes to perform both these two operations (write and update) in just one sequence. It has 2 different ways of accomplishing this: either write and update a selected DAC register or write to a selected register and update all the channels.

In the end we will only use the sequences that write and update the DAC registers, since in the context of this project, when we want to set a different voltage on a specific PMT, we want it to update immediately.

All the commands of DAC7568 are listed in annex 8.1. The figures show all the 32 bits of each sequence. Remember that in the case of DAC7568, the bits from DB7 to DB4, are always ignored.

4.3.3 The ADC MAX1240

General Description

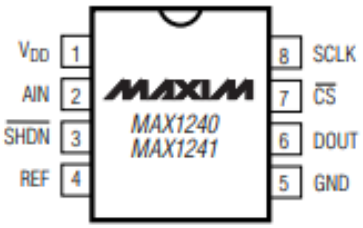

Figure 4.20. Pin configuration of the MAX1240.

The MAX1240, [20], is a low-power, 12-bit ADC that operates with a supply voltage in the range between 2.7 V to 3.6 V. It has a 3-wire serial interface that is compatible with SPI, QSPI and MICROWIRE. The device also features a 2 μA shutdown mode, that is controlled by pin 3 (SHDN) of Figure 4.20. Finally, the MAX1240 also has an internal reference of 2.5 V, that is enabled by setting the SHDN pin high, although it can also work with an external reference. The three pins of the serial interface are SCLK, CS and DOUT of Figure 4.20, and they must be connected to the SPI signals SCKL, to SS (or CS) and MISO, respectively. The DOUT pin sends a string of 12 bits via the MISO line, representing the voltage present in AIN. The power supply of 3.3 V will be connected to pin 1 (V_{DD}) and the digital ground to pin 5 (GND). The maximum clock speed is 2,1 MHz.

Conversion and serial interface

The MAX1240 can convert input signals in the range from 0 V to V_{REF}, taking approximately 9 μs (including the track/hold acquisition time).

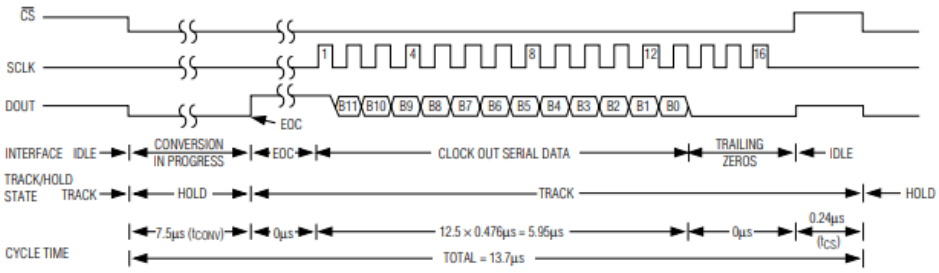

Figure 4.21. Serial interface timing sequence.

The conversion operation, which can be seen in Figure 4.21, starts by lowering the CS pin. After the specified 9 µs, conversion is complete and DOUT will be set high, signalling the end of conversion. By using external clock impulses, the 12-bit string is sent to the microprocessor via the MISO line (that must be connected to DOUT). Care must be taken to not generate clock impulses while the ADC is converting values, so the specified time must be respected before generating the clock signal. The maximum frequency of the clock is 2,1 MHz, and when using an SPI interface, CPOL and CPHA must be set to '0', which is the spi mode 00 (section 4.2).

Since the digital output is a 12-bit string and there is a bit signalling the end of conversion, 13 bits need to be clocked out of the ADC, although the first bit will have to be erased in order to get the correct digital voltage.

4.3.4 The bus switch SN74CB3Q3245

The SN74CB3Q3245 [21] is an 8-bit switch with a single output-enable input ($\overline{OE}$). On each side of the device there is two groups of ports: A and B (Figure 4.22). When the output-enable input is set low, the ports A are connected to the ports B, allowing bidirectional data flow between each other. When the output-enable input is set high, the switch is off and ports A and B are disconnected. The power-supply of the device is pin 20 (V_{CC}) and it accepts 3.3 V. On pin 10 (GND) a digital ground must be connected.

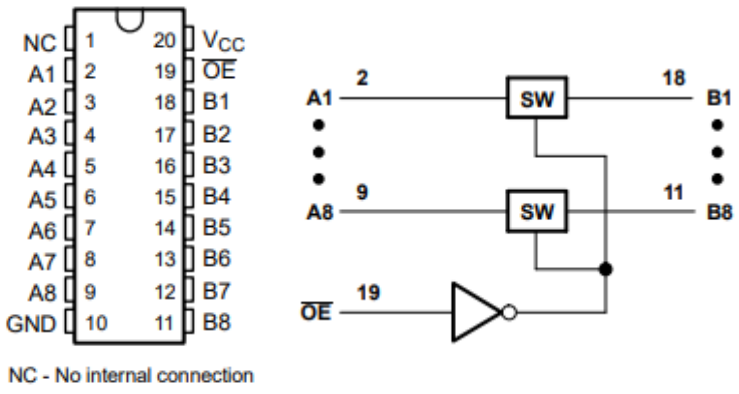

Figure 4.22. Pin configuration on the left, and functional block diagram of the bus switch on the right.

4.3.5 The MUX36S16

The MUX36S16, Figure 4.23, is an analogue multiplexer [22]. This device has 16 input ports, whose connection to its output port is controlled by a 4-bit address word. In this sense this is a 16:1 multiplexer with single-ended channels. The MUX36S16 can work either with dual supplies, between the ±5 V and the ±18 V or single supplies, between the 10 V and the 36 V. The power supplies can also be either symmetric (V_{DD}=V_{SS}) or asymmetric (V_{DD}≠V_{SS}).

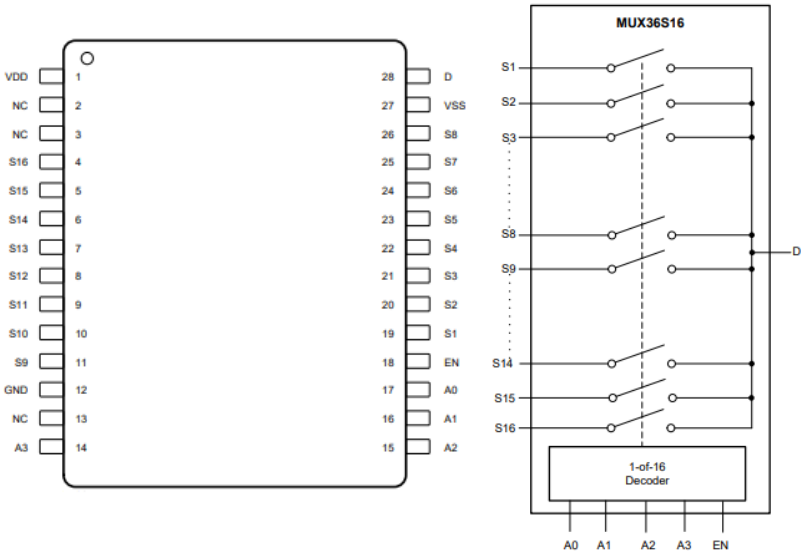

Figure 4.23. Pin configuration of the left and functional block diagram on the right.

Pin 1 (V_{DD}) is the most positive power-supply potential and pin 27 (V_{SS}) is the most negative power-supply potential. In the case of single-supply applications, pin 27 can be connected to ground (GND pin or pin 12). Pins 19 to 26 and 10 to 4 (S1 to S16) are the analogue inputs of the multiplexer. These are selected by pins 14, 15, 16 and 17 (A3, A2, A1 and A0) which are address lines that form a 4-bit word. The output of this MUX is on pin 28 (D). MUX36S16 offers an enable/disable feature controlled by the digital pin 18 (EN), that when it is set high, the device is enabled, and the analogue inputs can be selected by the digital address lines. When EN is set low, all analogue input's switches are turned off.

The pins 2, 3 and 13 (NC) are to be left unconnected.

4.4 Functional description of the digital control of HV Remote

As was explained in section 4.3.1, all HV Remote boards will share the same SPI lines. What will allow to communicate with only one board at a time is the digital bus switch SN74CB3Q3245, shown in the centre of Figure 4.24. This component will always be the first chip with whom the FPGA will communicate. To each HV Remote board will be associated a chip select signal (CS_CARD), that will be controlled by the FPGAs. This signal will be connected to pin 19 of the bus switch, which corresponds to the output-enable input ($\overline{OE}$). When we intend to communicate with a specific board, we instruct the FPGA to set its respective CS_CARD signal into a low state, thus making the bus switch connect its A ports to its B ports. This will link the SPI signals SCLK, MOSI and the 4 port expander's CS signals CS_SP1, CS_SP2, CS_SP3 and CS_SP4, coming from the FPGA to the HV Remote's inputs SCLK, DIN, CS_SPA, CS_SPB, CS_SPC and CS_SPD, respectively. The SPI signal MISO will be linked to the HV Remote output DOUT. The EXP_OUT is another output that is used as a test signal of the port expanders and can be one of these: DOUTA, DOUTB, DOUTC or DOUTD. Only one port expander can be tested at a time.

The 3.3 voltage supply is distributed by signal V3 and the common digital ground by signal DGND.

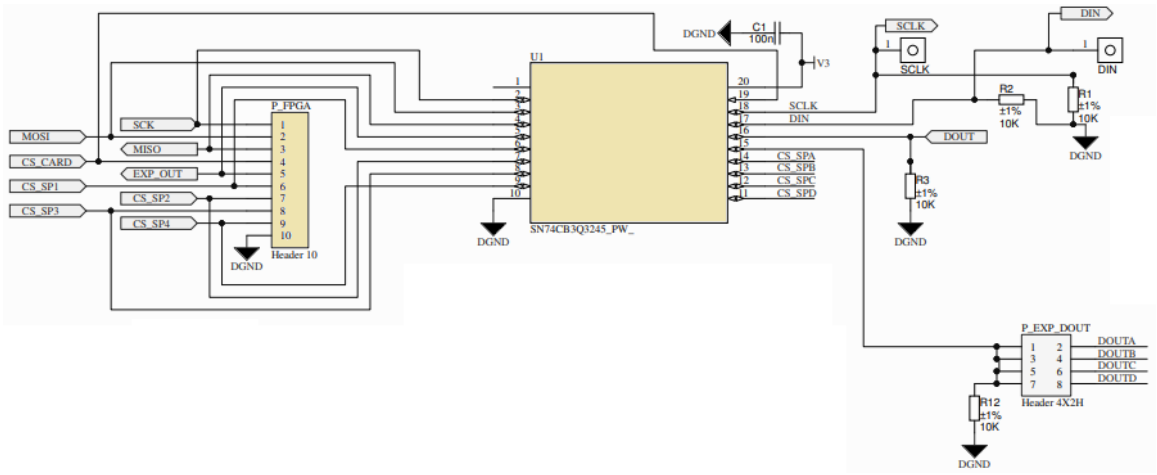

Figure 4.24. Inputs (SCLK, DIN, CS_CARD, CS_SPA, CS_SPB, CS_SPC and CS_SPD) and outputs (DOUT and EXP_OUT) of the HV Remote. In the centre the 8-bit bus switch is shown, connected to the FPGA on its left. The black filled triangles are the digital grounds of the HV Remote.

After enabling SPI communications for a specific HV Remote, the FPGA communicates with the port expanders to perform operations on the board. The port expanders, MCP23S17 (section 4.3.1), are the core of the HV Remote's digital control, since they allow the user to individually enable/disable the PMT's channels and control the CS signals of the other chips (the SYNC signals of the DACs and the ADC's CS).

Because there are 48 channels in each HV Remote, and the MCP23S17 has 16 ports, three of them (identified as port expanders A, B and C) will be used for the enable/disable functionality. Port expander D, Figure 4.25, will have the DAC and ADC chip-selects, and the MUXs' enables (which are the signals that activate the MUX's operation). Since there are enough free ports in this last expander, the MUXs' address codes were also assigned to the remainder of its GPIOs. The board's SPI signals SCLK and MOSI and the DOUTD signal are connected to pins 12, 13 and 14, respectively.

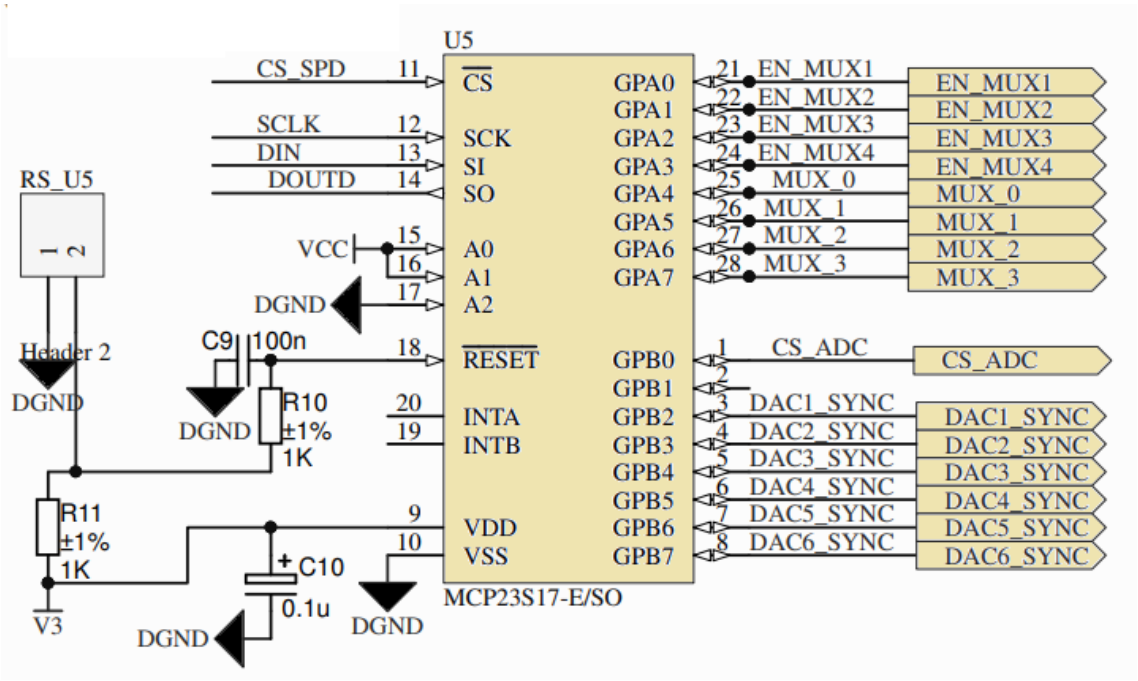

Figure 4.25. Port Expander D connections. On the bottom left corner is the voltage supply V3 that supplies the port expanders with 3.3 V. The address code of this expander is 011, with A2 connected to DGND and A1 and A0 connected to 3.3 V. Pins 19 and 20 are left unconnected.

On port expander D, the GPIOs from GPA0 to GPA3 (physical pins from pin 21 to pin 2) were connected to the multiplexer's enable signals (EN_MUX1, EN_MUX2, EN_MUX3 and EN_MUX4). The GPIOs from GPA4 to GPA7 (physical pins 25 to 28) represent the mux's address code that allows to select one of its 16 ports (MUX_3, MUX_2, MUX_1, MUX_0). These lines are shared by all multiplexers, but this is not a concerning issue since only one will be enabled at a time. So, for example if we want to adjust the voltage of a channel that is associated to MUX1, we must first write to port expander D to set its GPA0 as '1'. Then we must write the 4-bit word, which is associated to the desired channel, by setting the appropriate GPIOs state to high or low.

The GPB0 (pin 1) is connected to the chip-select of the ADC (CS_ADC). Whenever a reading is to be performed this GPIO port must be set to '0' and when the ADC is not operating it must be set to '1'.

The ports from GPB2 to GPB7 (pins 3 to 8) will be connected to the SYNC signals of the DACs (DAC1_SYNC, DAC2_SYNC, DAC3_SYNC, DAC4_SYNC, DAC5_SYNC, DAC6_SYNC). Every time we need to adjust a PMT's voltage we need to set low the respective DAC's SYNC signal which means we will have to change its respective GPIO to state '0' in order to be able to communicate with the DAC. After communicating with the DAC, the GPIO's state is set back to '1'.

The rest of the port expanders (A, B and C) have its GPIOs connected to the enable signals of the PMTs (CHi_ENj with i ranging from 1 to 4 and j from 1 to 12). So, if we need to disable/enable a specific photomultiplier, we will have to command a specific port expander to set the respective GPIO to '0'/'1'.

To each port expander was assigned a unique address (pins 15 to 17), although this functionality will not be used, for now. This is the reason why each one has a specific chip-select signal. Nevertheless, the device requires that its address pins always be externally biased.

The voltage supplies, VDD (pin 9), are connected to V3 and the VSS (pin 9) are connected to the common digital ground of the HV Remote board, DGND. The RESET (pin 18), which is a hardware reset feature, is positively biased with 3.3 V, as demanded by the device's specifications (see section 4.3.1). The connections to port expander A and D are shown in annex 8.2.

Because of the replacement of the analogue components by quadrupole versions in order to minimize the size of the HV Remote boards, the channels were organized in twelve groups of four channels each, hence the nomenclature CHi_ENj, CHi_CTRj and RDi_CHj, with i going from 1 to 4 and j from 1 to 12.

The DAC7568 will supply the regulation loop with the signal that will regulate the HV of the PMTs (CHi_CTRj). As shown in Figure 4.26, pins 3 to 6 and pins 8 to 11 are the analogue voltages that will be used for the HV regulation of the PMTs channels. Pin 1 corresponds to the signals DAC1_SYNC, DAC2_SYNC, DAC3_SYNC, DAC4_SYNC, DAC5_SYNC AND DAC6_SYNC of the 6 DACs, that are connected to the GPIOs of port expander D. Pins 14 (SCLK) and 13 (DIN) are inputs of the HV Remote, that are linked to the SPI signals SCLK and MOSI, respectively. The pin 7 corresponds to the DAC7568 V_{REF}IN/ V_{REF}OUT pin that will be left unconnected since we will use the DAC's internal reference, although a capacitor will be placed between this pin and DGND due to datasheet recommendations. DAC1 connections may be consulted in annex 8.3. The others aren't shown because they have the same kind of connections, only differing in the channels' numbers.

If, for example, we want to change the voltage of PMT1 (see annex 8.5), which is regulated by the CH1_CTR1 signal of Figure 4.26, we need to lower the DAC1_SYNC line of DAC 1. This line is connected to GPB2 of the port expander D (Figure 4.25).

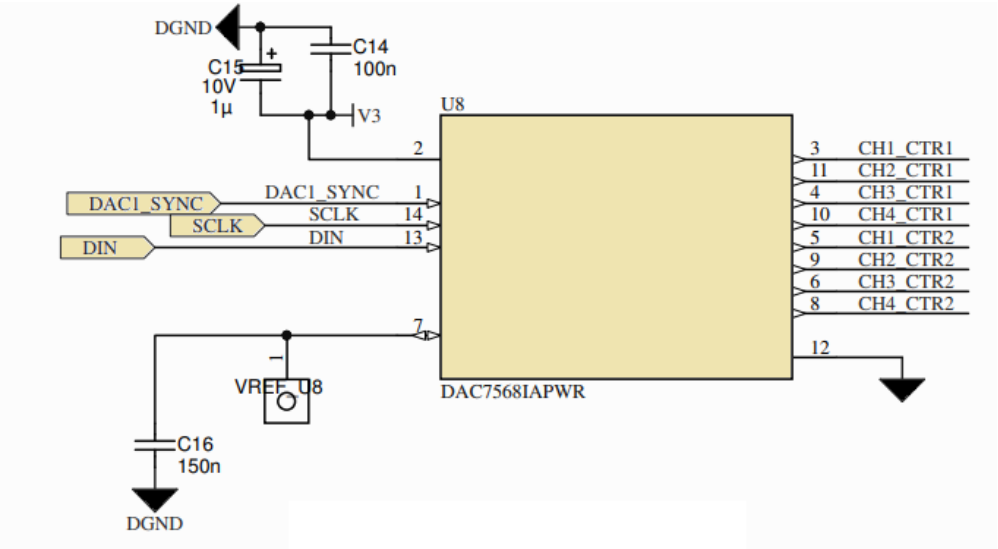

Figure 4.26. Connections of DAC1. The capacitors in the reference pin (pin 7) and in the power supply are recommendations from the datasheets. Pin 12 is the ground of the DAC which is connected to the common digital ground DGND.

So, in order to lower it, the CS_SPD signal must be lowered first, so that we can communicate with port expander D and make the necessary changes on its GPIO ports. Then we write in its GPIOB register in order to set GPB2 to a low state. After this, we close communications with expander D (by setting CS_SPD high) and write the specific "write and update" sequence of annex 8.1 to DAC1, in order to set its channel A (pin 3 of Figure 4.26) to the desired voltage. In the end, we must communicate again with port expander D, to set the respective SYNC line to a high state.

The MAX1240 performs all the voltage reading operations, which includes reading the PMT's voltages, the temperature probes and the test voltage. The connections of the ADC are shown in Figure 4.27. Pin 1 is connected to the voltage supply V3 and pin 5 is connected to DGND. The HV Remote's output DOUT and the SCLK signal are connected to pin 6 and 8, respectively. The chip-select of the ADC, CS_ADC, is connected to port expander D's GPB0. The SHDN pin of MAX1240 (pin 3 in Figure 4.27) is connected to V3, in order to always stay positively biased, so that the internal reference is enabled. Some capacitors are also connected between the voltage supply and DGND, and between the internal reference pin and DGND, for stability purposes.

Whenever a reading is to be made, GPB0 (connected to CS_ADC) of expander D must be lowered. Then, MAX1240 starts conversion of AIN, and a small delay time must be set, before the clock starts to be generated (see section 4.3.3). When SCLK starts, the ADC will send a 12-bit word to the FPGA, via the DOUT line, indicating the value of the voltage in pin AIN (that must be later converted to volts).

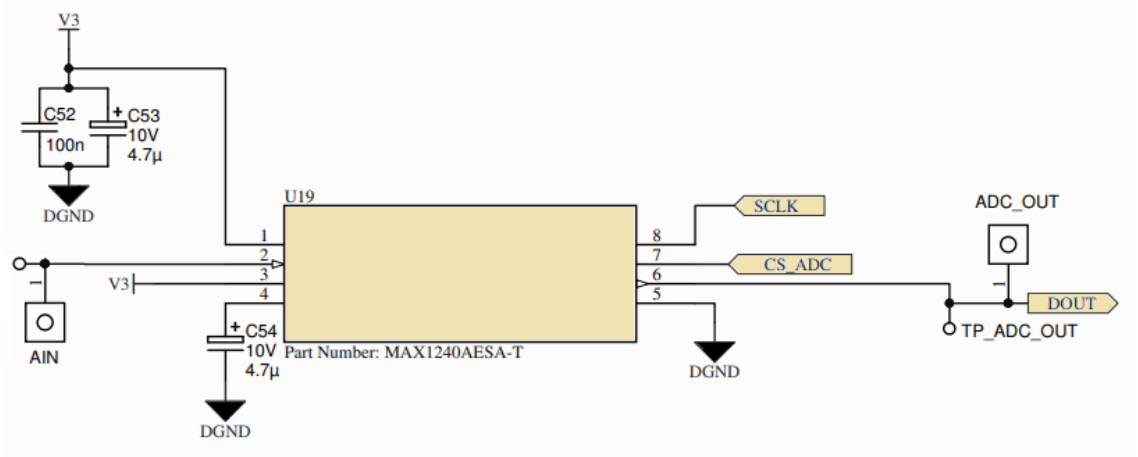

Figure 4.27. Connections of the ADC. AIN is the analogue input voltage that is selected by the multiplexers. After reading and converting the analogue voltage to a 12-bit digital word, the ADC sends it to the master device (FPGA) via the DOUT line that is linked to the SPI MISO line.

To connect analogue outputs to the AIN pin there is 4 multiplexers named MUX1, MUX2, MUX3 and MUX4. MUX1 is shown on Figure 4.28. All MUXs share the MUX_0, MUX_1, MUX_2 and MUX_3 signals which are the address lines, that will be connected to GPA4, GPA5, GPA6 and GPA7 of port expander D, respectively. Their enable signals (EN_MUX1, EN_MUX2, EN_MUX3 and EN_MUX4), also controlled by expander D, allow us to have only one active MUX at a time. Their outputs also share the same line (OUT_MUXs), which is connected to the AIN pin. All MUXs will be supplied by analogue voltages VP12 and VN12, which correspond to +12 V and -12 V. Decoupling capacitors are also connected for reliable operation. Pin 12 is connected to the analogue ground of HV Remote (AGND). Pins 4 to 11 and 19 to 26 of MUX1

and MUX3 are connected to the RDi_CHj signals (with i ranging from 1 to 4 and j from 1 to 4 for MUX1, and from 7 to 10 for MUX3) which are the PMTs channels' voltage reading signals. MUX2 has pins 19 to 26 connected to RDi_CHj signals, where i goes from 1 to 4 and j from 5 to 6. Its pin 11 is connected to the test voltage (TEST) and pins 10 and 9 are connected to the temperature probes signals (TEMP1 and TEMP2). MUX4 has the remaining RDi_CHj channels, which are the RDi_CH11 and RDi_CH12, with i ranging from 1 to 4.

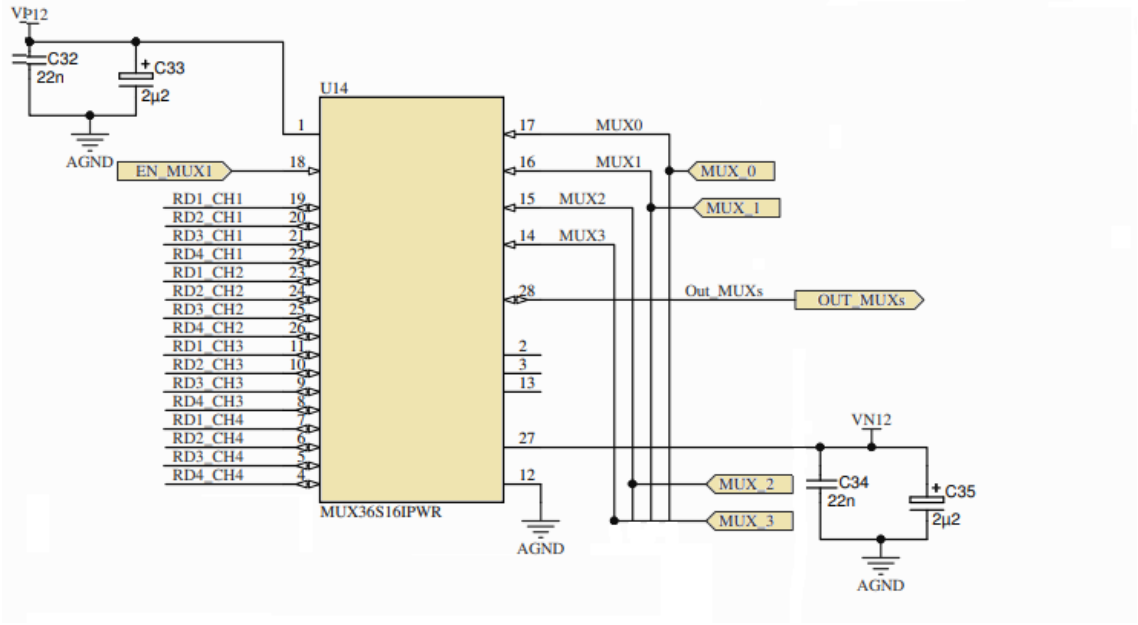

Figure 4.28. Connections of MUX1. Pins 2, 3 and 13 are left unconnected. VP12 and VN12 are the analogue voltage supplies ±12 V of the HV Remote.

All reading operations start by setting high the respective MUX's EN, and then by configuring its address lines in order to select one of its input channels. To accomplish this, we must first enable communications with the port expander D in order to configure its GPA ports which are the ones that are connected to the MUX's EN signals and their address lines. After having these ports configured, we can proceed to communicate with the ADC to get a reading from it.

The connections of MUX1 may be consulted on annex 8.4. The other MUXs have almost the same kind of connections, so they aren't shown.

5 The HV Remote boards Tester

5.1 General Description of the tester

Since 256 boards will be produced to supply all the TileCal PMTs, an automated tester will be needed to check the functionality of these boards, before being sent to CERN.

This tester will mainly perform functional tests to the boards: reading channels' voltages, enable/disable channels, set the channels' voltage and read temperature signals. Eventually some tests at higher temperatures will also be done, in order to evaluate how the boards, behave at different temperature regimes.

The tester will need a software interface in order to establish the SPI communication with the board. This interface will be written using the version 3 of the Python language and will be developed using a Raspberry Pi 3b+ which was the system that was recommended by CERN, and that had the sufficient performance parameters to test the HV Remote boards. The code is intended to run in a Raspberry Pi 3b+ [28], which may be used directly as a computer by connecting it to a screen and using an SD Card with an operating system, or indirectly by controlling it remotely with a computer via SSH (this option requires the Raspberry to be connected to the computer with an ethernet cable or to be connected to the same wi-fi network). The Raspberry was chosen due to its high computational power, and large number of GPIOs (that will be needed to control the chip select of the port expanders of the HV Remote).

The Raspberry Pi 3b+ is designed with a SoC (System on Chip) denominated BCM2835 which contains a microprocessor ARM with several peripherals (GPIO, SPI, …) [29]. Some of the important features of the Pi will be explained in the next section.

After the completion of the software interface, a GUI (Graphical User Interface) will be designed to have a better interaction between the user and the Raspberry (which will run the software interface for the HV Remote boards), and a better control of the test results. These results will be displayed, in the future, in real-time plots, that will be updating the voltages of each PMT channel.

5.2 The Raspberry Pi 3b+

The Raspberry Pi is a series of small computers equipped with a Broadcom SoC (System on Chip), USB (Universal Serial Bus) and ethernet ports, and several GPIO pins which support communication protocols like the SPI. It also has a HDMI video output, and a jack for audio output. Some models support Wi-Fi and Bluetooth. The Raspberry Pi's OS (operative system) is stored in an SD (Secure Digital) card that must be inserted, before turning on the Rasp.

Model 3b+ has four USB ports, one ethernet port and 40 physical pins, that include 27 GPIO pins. It also has on-board Wi-Fi and Bluetooth and supports some features to boot the device without an SD card, although those won't be used. Its Broadcom SoC is a BCM2835 that includes a 700 MHz ARM processor, a VideoCore IV graphics processing unit (GPU) and a RAM (Random Access Memory).

The peripherals that may be accessible by the ARM processor are [29]:

- Timers
- Interrupt controller
- GPIO
- USB
- PCM / I2S
- DMA controller
- I2C master
- I2C / SPI slave
- SPI0, SPI1, SPI2
- PWM
- UART0, UART1

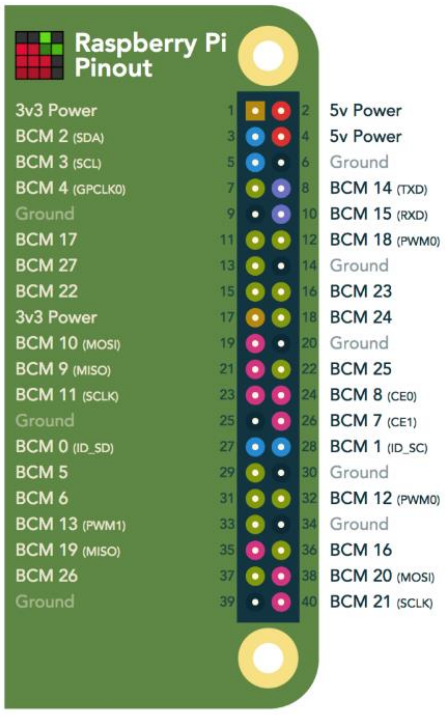

Figure 5.1. Raspberry Pi b+ pinout. The name of the pins is shown as BCMi, where i corresponds to the numbering of the GPIOs of BCM2835.

We will focus only on the SPI peripheral, since that is the one that will be used in this project.

The SPI peripheral allows the Raspberry to behave either as a slave or master. In master mode the peripheral implements the standard 3-wire serial protocol (MOSI, MISO, SCLK) and two independent chip-select signals CE0 and CE1. The SCLK wire can be generated with any of the 4 clock modes (see section 4.2), with data being always transmitted MSB first.

The SPI peripheral offers three SPI controllers but only one, SPI0, is available on the Raspberry Pi header (Figure 5.1) via one SPI bus. The SPI bus can be accessed via the pins 19 (MOSI), 21 (MISO) and 23 (SCLK), or via pins 35 (MISO), 38 (MOSI) and 40 (SCLK) of Figure 5.1, respectively. The SPI controller has two chip-select signals, the CE0 (pin 24) and CE1 (pin 26) that allow the Raspberry Pi to automatically control communications with the slaves. If we desire to perform this control manually, we can use the GPIO ports, by associating them to the slaves' chip-select pins.

Each SPI transaction occurs with a maximum of 8 bits. But we've seen that the HV Remote digital components communicate with longer words. For example, the port expander must always receive 16 bits for the communication to be successful, so the Raspberry Pi will have to perform two transactions of 8 bits each. The chip-select signals will be kept low for all the needed transactions for all the devices.

On Figure 5.1 we can see that the Raspberry Pi's GPIOs also have two 3.3 V pins, that will be useful to power the digital devices (the port expanders, the DACs and the ADC) during the tests. We will only use the SPI1 wires, since we only need one set of SPI lines, and the chip-select signals will be controlled manually via the GPIOs.

As was said previously, the interface will be written in Python 3, with an object-oriented methodology. There is an already developed python module named SpiDev, that was specifically designed to be used in the Raspberry systems to performs SPI transactions, although it is also compatible with other Linux distributions. This is a very intuitive and useful interface, because by using it we won't need to directly write in the SPI hardware, which would become a much more complex task.

The SpiDev module allows the user to create an SPI object by assigning a name to spidev.SpiDev(). This object exposes the following functions/methods:

- **open(bus, device)** – this connects the python object to the SPI controller, where bus is always 0, and device may be either 0 or 1 (which corresponds to CE0 and CE1, respectively).
- **close()** – disconnects the object from the SPI controller.
- **bits_per_word** – this sets the number of bits that we desire to send in each word. The number of bits in each transaction in the Raspberry Pi is limited to 8 bits, so this function will not work in the Pi, although it may be compatible in other Linux systems.
- **cshigh** – a property that gets/sets if the chip-select signal is active high.
- **loop** – property that gets/sets "loopback configuration". This is useful for test purposes.
- **lsbfirst** – property that gets/sets if LSB should be transferred first or last. The Raspberry Pi can only send word with MSB first, so to send information with LSB first, we need to make some changes to the word manually before sending it to a slave device.
- **max_speed_hz** – this property gets/sets the maximum clock speed in Hz.
- **mode** – this gets/sets the SPI mode with a two bits code ([CPOL | CPHA]). The available codes correspond to the ones shown in Table 4.1.
- **threewire** – this is a property used to set a SPI communication with only three lines, where the MISO and MOSI signals are shared by the same line. This functionality wasn't used because a functionality of the HV Remote would be lost.
- **readbytes(len)** – this function takes as an argument an integer number (len), that specifies the number of bytes that are to be read via the MISO line, ignoring what is sent via the MOSI line.
- **writebytes([values])** – this function takes as an argument an array of values that may be either a number or a character, that specifies the information that is to be sent via the MOSI line, ignoring what is read via the MISO line. Each element of the array corresponds to one transaction of a byte.
- **xfer([values])** – this function performs an entire SPI transaction (performs a write operation via the MOSI line and stores the value read via the MISO line). Takes as arguments an array of either numbers or characters and reads the same number of bytes as

the ones that were sent. This function causes the CE signal that is being used to be released and reactivated between byte transactions.

- **xfer2([values])** – this has the same function as xfer, but the CE signal will be held active during the whole communication.

The most relevant functions/methods for the software interface will be the xfer2() and the max_speed_hz, and obviously the open(bus, device) and close() methods. An important note on the max_speed_hz property, is that it only works with a restricted number of clock speeds available. So, although this property accepts any input number, the actual speed of the clock will be set to the greatest speed inferior to the one specified by the user. The clock SPI speed is obtained by dividing the system's clock (Core Clock) which is 250 MHz, by a clock divider (CDIV), as shown in equation (6.1.

$$\mathrm{SCLK} = \frac{\mathrm{Core\ Clock}}{\mathrm{CDIV}} \tag{5.1}$$

All the possible SPI clock speeds are shown in Table 5.1. If for example we set the max clock speed as 10 MHz, the SPI clock will be set to 7.8 MHz.

Table 5.1. SPI clock speeds.

CDIV	SCLK
2	125.0 MHz
4	62.5 MHz
8	31.2 MHz
16	15.6 MHz
32	7.8 MHz
64	3.9 MHz
128	1953 kHz
256	976 kHz
512	488 kHz
1024	244 kHz
2048	122 kHz
4096	61 kHz
8192	30.5 kHz
16384	15.2 kHz
32768	7629 Hz

To familiarize with the SpiDev class of the Raspberry Pi, a preliminary test was made with an Arduino Uno acting as a slave. The Raspberry's SPI GPIO pins were connected to the respective Arduino Uno's SPI GPIOs. Code emulating the Raspberry as a SPI master, was written and is shown in Figure 5.2. This code uses a string stored in the variable test_str or an integer stored in the variable num, to send it to the Arduino by using the SpiDev xfer2 method. Before sending any information (either the string or the integer), the function encode_bytes() is used to encode the

argument into an array of bytes. This array is stored in the variable test_lst. Then we call method xfer2, by taking as its argument test_lst and assign the return value of this method to the variable reply. In the end the variable reply is decoded by a similar function called decode_bytes, and the decoded value is printed on the screen. In the case of **Erro! A origem da referência não foi encontrada.** the value that chosen to test_str was the word "alpha" ("\n" is interpreted as a single paragraph). If the user uses the shortcut Ctrl+C while the program is running, a KeyboardInterrupt exception will be sent, stoping the program and closing the SPI connection to the SPI controller.

```
1   #!/usr/bin/python3
2
3   import spidev
4   import time
5   import libs.support as support
6
7   # Open SPI lines
8   spi=spidev.SpiDev()
9   spi.open(0,0)
10  spi.max_speed_hz=7629
11  spi.bits_per_word=8
12
13  DELAY=0.1
14
15  num=27819150
16  test_str="alpha\n"
17  test_lst=support.encode_bytes(test_str)
18
19  try :
20      while True:
21          reply=spi.xfer2(test_lst)
22          print("reply = ", reply, "decoded =", support.decode_bytes(reply))
23          time.sleep(DELAY)
24
25  except KeyboardInterrupt:
26      spi.close()
```

Figure 5.2. Raspberry Pi SPI test code. Highlighted by the rectangles are the initialization of the SpiDev object (blue), the connection to the SPI controller (orange), the definition of the max clock speed (green), the SPI transaction (red) and the closing of the SPI connection to the controller (white). A delay of 0.1 s is created and is used after each print, in the while cycle.

The code used for the Arduino Uno (that was connected to a computer) was based on the one written by Nick Gamon and available online [30]. What it does is essentially to configure the Arduino as a slave device, and access the Arduino's hardware directly, to get the bytes received and stored in the SPDR (SPI Data Register), and subsequently print them in the computer's screen in order to compare its value with the one that the Raspberry Pi receives back.

The test was useful to get to know how to work with a Raspberry and with SpiDev and its several methods. The words received by the Arduino and by the Raspberry always had the same characters, so the results were satisfactory.

5.3 The HV Remote software interface

Having familiarized with Raspberry Pi and the SpiDev module, we started to create a software interface that would allow us to communicate with the HV Remote boards. This was done by creating an HV Remote Python class with methods tailored to the necessary operations that the boards will need to perform (adjusting and reading voltages and enabling/disabling channels). To accomplish this, we first needed to create interfaces for the main digital components (the DAC7568, the ADC MAX1240 and the port expander MCP23S17), by creating a Python class for each one. The MUX36S16 and the 8-bit digital bus switch SN74CB3Q3245 don't need a software interface, because they don't have any kind of serial interface. The only things that we need to do in these components is to change the state of some channels: the output-enable input for the bus switch and the enable and address channels of the multiplexers. These channels will be controlled directly by the Raspberry Pi and the port expander, respectively.

From now on a description of all the components' interfaces will be presented, and in the end, a description of the HV Remote along with some tests applied to it will be shown.

5.3.1 The MCP23S17 class

This class was based on a class already existent but with some modifications that fit the requirements of the HV Remote project. This code was written by Florian Mueller and is available online [31]. The class has some constants, that were defined with uppercase letters. These include the MCP23S17 registers' addresses and some flags that help setting some register's bits in an easier way (for example when configuring the port expander). Each instance of this class must be initialized (the constructor of the class is shown in Figure 5.3) with four mandatory arguments: one instance of a SpiDev class, one instance of the Raspberry Pi's GPIO class[3], the Pi's pin number that is associated with the port expander's chip select and the port expander address code. The instance can also receive an optional argument, pin_reset, which would allow us to control the MCP23S17's reset hardware feature with the Pi, if the port expander's RESET pin is connected to one of the Raspberry's GPIOs.

```
def __init__(self, spiDevice=spidev.SpiDev(), device_id=0x00, pin_reset=-1, chip_select=-1, iGPIO=GPIO):
    """
    Constructor
    Initializes all attributes with 0.

    Keyword arguments:
    device_id -- The device ID of the component, i.e., the hardware address (default 0)
    pin_cs -- The pin of the Raspberry Pi that will be used as chip select of the MCP
    pin_reset -- The pin of the Raspberry Pi that will be used as reset of the MCP
    """
```

Figure 5.3. MCP23S17 class constructor. The arguments, in blue, represent the SpiDev instance (spiDevice), the port expander address code (device_id), the Pi's pin number associated with the port expander's RESET pin (pin_reset), the Pi's pin number associated with the port expander's CS pin (chip_select) and an instance of the RPi.GPIO class (iGPIO).

[3] The Raspberry Pi has a Python class, named RPi.GPIO that allows to control the GPIOs of the Pi by its GPIO numbering or by the numbering of the pins.

The constructor of this class grabs these arguments and stores them as attributes of the created instance, Figure 5.4. The attributes are the variables that are preceded by the "self." keyword, where the ones that have an additional "__" character are determined as private arguments, that can be accessible only in the methods of their own class, and the others are determined as public arguments, therefore being able to be accessed by any program. The constructor also sets the SPI clock mode that must be used when performing SPI transactions, with the attribute "__spiMode". The attributes "GPIOA", "GPIOB", "IODIRA", "IODIRB", "GPPUA" and "GPPUB" store the current values of the respective MCP23S17 registers. The attribute "isInititialized" will appear in all classes and was useful in the earlier design stages of the codes, to guarantee that methods were only used if the chips were properly configured.

```
94          self.device_id = device_id
95          self.__pin_reset = pin_reset
96          self.__chip_select = chip_select
97          self.__GPIO = iGPIO
98          self.__spiMode = 0b00
99          self.GPIOA = 0
100         self.GPIOB = 0
101         self.IODIRA = 0
102         self.IODIRB = 0
103         self.GPPUA = 0
104         self.GPPUB = 0
105         self.__spi = spiDevice
106         self.isInitialized = False
107
108         self.__setupGPIO()
109         self.isInitialized = True
110         self.__writeRegister(MCP23S17.MCP23S17_IOCON, MCP23S17.IOCON_INIT)
111
112         # set the pins as outputs
113         for index in range(21, 29):
114             self.setDirection(index, MCP23S17.DIR_OUTPUT)
115         for index in range(1, 9):
116             self.setDirection(index, MCP23S17.DIR_OUTPUT)
```

Figure 5.4. Content of the MCP23S17 constructor (the method __init__(self, spiDevice=spidev.SpiDev(), device_id=0x00, pin_reset=-1, chip_select=-1, iGPIO = GPIO)).

In the end of the constructor, the MCP23S17 and the Raspberry Pi's GPIOs are configured. The private method "__setupGPIO()", configures the attributes "__pin_reset" and "__chip_select" as Pi's outputs. The private method of line 110 of the code of Figure 5.4 is the one that will be responsible to do all SPI transactions with the MCP23S17, and its content is shown in Figure 5.5. It takes as arguments the register's address that we intend to communicate with, and the value that we wish to write. In this line we are writing the binary value 00101000 to the IOCON register. This will set IOCON's SEQOP and HAEN bit to '1', thus disabling sequential operation and enabling the address pins[4] (see section 4.3.1).

[4] Although we said that the HV Remote won't use this feature for now, we enable it anyway, to afterwards make some comparisons when using the address pins and only one chip-select or using several chip-selects.

In the beginning the "__writeRegister" method asserts if the port expander was properly configured. If it was, the attribute "isInitialized" will have the Boolean value "True", therefore passing the assertion. The "command" variable represents the control byte, which determines if it is a read or write operation and contains the port expander's address code. Then the SPI mode is set to 00 (CPOL = 0 and CPHA = 0) by the "__setupSPI()" method, that also adds a small delay to let the SCLK signal to correct itself. The Raspberry Pi's GPIO that is connected to the port expander's chip-select is set low on the fifth line of the code, thus enabling SPI communication. The transaction is made by the SpiDev's xfer2 function, by sending 3 bytes: the control byte ("command"), the register's address ("register") and finally the word that we want to write ("value"). In the end the chip-select is brought high again, disabling the communication with the device.

```
285     def __writeRegister(self, register, value):
286         assert self.isInitialized
287         command = MCP23S17.MCP23S17_CMD_WRITE | (self.device_id << 1)
288         self.__setupSPI()
289         self.__GPIO.output(self.__chip_select, self.__GPIO.LOW)
290         self.__spi.xfer2([command, register, value])
291         self.__GPIO.output(self.__chip_select, self.__GPIO.HIGH)
```

Figure 5.5. This is the "__writeRegister" private method. The first line of code defines the name of the method and the arguments it must be given. The second line asserts if the flag "isInitialized" has the Boolean value "True". The SPI transaction is performed by the "__spi.xfer2" line.

The last lines of Figure 5.5 code (line 114 and 116) have a method named "setDirection" that takes as arguments a pin number of the MCP23S17 GPIOs and one of the two MCP23S17 class constants DIR_OUTPUT and DIR_INPUT, that are '1' and '0', respectively. This method defines if the pin "index" will be an output or an input, by modifying the respective bit of the IODIR register, without altering the other bits. In the last lines of the class constructor, we are configuring all GPIOs as outputs with a for cycle.

The content of the "setDirection" public method is shown in Figure 5.6. Lines 177, 178 and 179 do all the necessary assertions for the method to work properly. Line 177 asserts if the "pin" argument is valid (it must be in the interval from 1 to 9 or from 21 to 29), line 178 asserts if the "direction" value is either DIR_INPUT or DIR_OUTPUT and line 179 asserts if initialization was made correctly. Then an if-else clause separates the situations where "pin" is in the interval 1-8, which corresponds to the GPIOB register, or 21-28, which is the GPIOA register. In each situation, the code stores the values of the attributes "IODIRA" and "IODIRB" into a local variable named "data" and defines the value of the variable "register" which will contain the address of the target register address. The variable "noshifts" that is defined in the two situations indicates the bit of the IODIR register that is to be changed.

Then another if-else clause alters the needed bit of the "data" value, according to the value of the parameter "direction". If by chance, "direction" is DIR_INPUT, the bit will be changed to '1', else it will be changed to '0'. The SPI transaction is performed in line 195.

In the end, the attributes of the registers are updated to the new values.

```
169     def setDirection(self, pin, direction):
170         """Sets the direction for a given pin.
171 
172         Parameters:
173         pin -- The pin index (0 - 15)
174         direction -- The direction of the pin (MCP23S17.DIR_INPUT, MCP23S17.DIR_OUTPUT)
175         """
176 
177         assert pin in range(1, 9) or pin in range(21, 29)
178         assert direction == MCP23S17.DIR_INPUT or direction == MCP23S17.DIR_OUTPUT
179         assert self.isInitialized
180 
181         if pin in range(21, 29):
182             register = MCP23S17.MCP23S17_IODIRA
183             data = self.IODIRA
184             noshifts = pin - 21
185         else:
186             register = MCP23S17.MCP23S17_IODIRB
187             noshifts = (pin + 7) & 0x07
188             data = self.IODIRB
189 
190         if direction == MCP23S17.DIR_INPUT:
191             data |= (1 << noshifts)
192         else:
193             data &= (~(1 << noshifts))
194 
195         self.__writeRegister(register, data)
196 
197         if pin in range(21, 29):
198             self.IODIRA = data
199         else:
200             self.IODIRB = data
```

Figure 5.6. Content of the "setDirection" public method. Takes in as argument a pin number of the MCP23S17 GPIOs and one of the values DIR_INPUT and DIR_OUTPUT. The acceptable numbers for the "pin" argument range from 1 to 8 (GPBs) and from 21 to 28 (GPAs).

The methods that the MCP23S17 class offers are shown in Figure 5.7. Some were already seen because they are used in the class' constructor and others have a very similar structure to the ones that were mentioned. For example, the "__readRegister" method is equal to its write equivalent, the only difference being the "command" variable. The "setPullupMode" is also equal to the "setDirection" method, the difference being that the register with which we communicate is the GPPU instead of IODIR.

The "__writeRegisterWord" and "__readRegisterWord" are methods that write/read to both bank A and bank B registers by calling two times the "__writeRegister" / "__readRegister" method.

The "reset" method is not being used now, but it was written in order to see if we could control the MCP23S17 RESET pin by software. The method also resets the attributes values to 0. The "digitalWrite" and "writeGPIO" methods will now be explained since they are the most important. The read equivalent methods won't be explained since they are the same as the write ones, except for the methods that they call. The "readGPIO" and "digitalRead" call the "__readRegisterWord" and "__readRegister" methods, while the "writeGPIO" and "digitalWrite" call the "__writeRegisterWord" and "__writeRegister", respectively.

```
84  >   def __init__(self, spiDevice=spidev.SpiDev(), device_id=0x00, pin_reset=-1, chip_select=-1, iGPIO=GPIO): ···
117
118 >   def reset(self): ···
135
136 >   def setPullupMode(self, pin, mode): ···
168
169 >   def setDirection(self, pin, direction): ···
201
202 >   def digitalRead(self, pin): ···
229
230 >   def digitalWrite(self, pin, level): ···
261
262 >   def writeGPIO(self, data): ···
273
274 >   def readGPIO(self): ···
284
285 >   def __writeRegister(self, register, value): ···
292
293 >   def __readRegister(self, register): ···
301
302 >   def __readRegisterWord(self, register): ···
308
309 >   def __writeRegisterWord(self, register, data): ···
313
314 >   def __setupGPIO(self):        ···
320
321 >   def __setupSPI(self): ···
```

Figure 5.7. MCP23S17 class methods. The ones that are preceded by the underscore character, "__", are private methods, and the others are public methods.

The "writeGPIO", Figure 5.8, takes in as argument a 16-bit value, where the first 8 bits will be the new value for the GPIOA register and the second sequence of 8 bits will be the value to write in the GPIOB register. Therefore, this method allows us to write a value in all the port expander's GPIOs with just a single command. To perform the SPI transaction, this method calls the private method "__writeRegisterWord", that takes in as argument the address of a bank A register and the value that is to be written in both bank A and bank B respective registers.

```
262     def writeGPIO(self, data):
263         """Sets the data port value for all pins.
264         Parameters:
265         data - The 16-bit value to be set.
266         """
267
268         assert self.isInitialized
269
270         self.GPIOA = (data & 0xFF)
271         self.GPIOB = (data >> 8)
272         self.__writeRegisterWord(MCP23S17.MCP23S17_GPIOA, data)
```

Figure 5.8. Content of the "writeGPIO" method. It takes in as argument a 16-bit value ("data").

The “digitalWrite” method, Figure 5.9, allows us to change the value of only one GPIO of the port expander, by giving the method two arguments: the pin number of a specific GPIO port and the digital level we wish to set that GPIO (either high ‘1’, or low ‘0’). From 1 to 8 are the GPIOB pins and from 21 to 28 are the GPIOA pins. The “level” argument can be chosen from one of the class constants LEVEL_LOW or LEVEL_HIGH. This method does not change the state of the other GPIO ports, because it stores their value (which is obtained from the attributes “GPIOA” or “GPIOB”) on the local variable “data”. As with the “setDirection” method the variable “noshifts” determines the position of the bit that must be changed. In the end the code updates the values of the attributes.

```
230     def digitalWrite(self, pin, level):
231         """Sets the level of a given pin.
232         Parameters:
233         pin -- The pin index (1 - 9 or 21 - 29)
234         level -- The logical level to be set (LEVEL_LOW, LEVEL_HIGH)
235         """
236 
237         assert self.isInitialized
238         assert pin in range(1, 9) or pin in range(21, 29)
239         assert (level == MCP23S17.LEVEL_HIGH) or (level == MCP23S17.LEVEL_LOW)
240 
241         if pin in range(21, 29):
242             register = MCP23S17.MCP23S17_GPIOA
243             data = self.GPIOA
244             noshifts = pin - 21
245         else:
246             register = MCP23S17.MCP23S17_GPIOB
247             noshifts = (pin + 7) & 0x07
248             data = self.GPIOB
249 
250         if level == MCP23S17.LEVEL_HIGH:
251             data |= (1 << noshifts)
252         else:
253             data &= (~(1 << noshifts))
254 
255         self.__writeRegister(register, data)
256 
257         if pin in range(21, 29):
258             self.GPIOA = data
259         else:
260             self.GPIOB = data
```

Figure 5.9. Content of the “digitalWrite” method. Takes in as arguments a pin number of the MCP23S17 GPIOs, and one of the class’ constants LEVEL_HIGH or LEVEL_LOW that correspond to ‘1’ and ‘0’ respectively.

5.3.2 The DAC7568 class

The DAC class was built from zero, and all the DAC7568 functionalities, like the power up/down channels, enable/disable internal reference, switch internal mode and the different write modes (see section 4.3.2) were implemented. Each instance of this class must be initialized with the following arguments (Figure 5.10): a SpiDev instance, a MCP23S17 instance (which represents the port expander D, that will control the SYNC signal of the DACs) and the number of MCP23S17's pin that will be assigned to the SYNC of the DAC.

```
24      def __init__(self, spiDevice = spidev.SpiDev(), SYNC = -1, MCP = MCP23S17):
25          """
26          Constructor
27
28          Keyword Arguments:
29          SYNC -- pin of the MCP23S17 that will be used as SYNC (GPB2 - GPB7 or pin3 - pin8).
30                  The write sequence starts by bringing SYNC low
31          MCP -- the port expander that will control the SYNC pin of the DAC
32          mode -- mode of communication with the Internal Reference. There is two possible modes:
33                  'static' (which is the default) and 'flexible'. When using the flexible mode,
34                  the static mode is disabled and the mode must be switched first.
35          flexMeth -- in flexible mode there are 2 methods for enabling the internal reference.
36                          In method 1, the internal reference powers down if all DAC channels
37                          power down, and powers up if any DAC channel powers up.
38                          In method 2 the internal reference is powered up and stays in that
39                          state regardless of the state of the DACs
40          intRef -- indicates if the internal reference is enabled or disabled
41          chData -- dictionary that stores the values in each channel of the DAC
42          chState -- dictionary that points out if the channels are powered on or powered off
43                      in one of the three modes: '1k' (1k resistance to GND), '100k'
44                      (100k resistance to GND) or 'HIGH-Z' (high impedance to GND)
45          """
```

Figure 5.10. DAC7568 class constructor. Each instance must receive as arguments a SpiDev instance ("spiDevice"), a MCP23S17 instance ("MCP") and a MCP23S17 pin number (SYNC).

These arguments will be assigned to attributes as was done in the MCP23S17 class. Five additional attributes. The "__spiMode" defines the SPI clock mode for the DAC, which must set CPOL=0 and CPHA=1. The "__mode" is a flag indicating which is the current internal reference mode ('static' or 'flexible'). The "__flexMeth" specifies which is the method of the flexible mode that was chosen to enable the internal reference (either 1 or 2 and 0 when the internal reference's mode isn't 'flexible'). The "__intRef" takes a Boolean value ("True" or "False") to indicate if the internal reference is enabled or disabled. Finally, two Python dictionaries are created ("__chData" and "__chState") that give information of the value at the state (power-down or power-up) of each channel of the DAC. All these additional attributes start with default values.

In the end of the constructor, the flag "__isInitialized" is set to "True" after the necessary GPIOs of the port expander "MCP" have been properly configured by the private method "__setupGPIO()".

```
46          self.__spi = spiDevice
47          self.__spiMode = 0b10
48          self.__SYNC = SYNC
49          self.__MCP = MCP
50          self.__mode = 'static'
51          self.__flexMeth = 0
52          self.__intRef = False
53          self.__chData = {
54              'A': 0,
55              'B': 0,
56              'C': 0,
57              'D': 0,
58              'E': 0,
59              'F': 0,
60              'G': 0,
61              'H': 0
62          }
63          self.__chState = {
64              'A': 'On',
65              'B': 'On',
66              'C': 'On',
67              'D': 'On',
68              'E': 'On',
69              'F': 'On',
70              'G': 'On',
71              'H': 'On'
72          }
73          self.__isInitialized = False
74          self.__setupGPIO()
75          self.__isInitialized = True
```

Figure 5.11. Content of the DAC7568 constructor. It assigns the instance's arguments to attributes and creates additional attributes that act as flags to the remainder of the program or indicators of the state of the DAC's registers (DAC channel's register, internal reference's register and power-down logic's registers).

This class offers a number of different methods, shown in Figure 5.12, that perform all the possible operations of the DAC7568 device. In order to adjust the DAC7568's channels voltages the "writeAndUpdateChanel()" method was the only one of the different write methods used. The other method that was also used was the "enableReference()" because we are going to use the internal reference of the DAC. The "__sendWord()" method, Figure 5.13, is the one that performs all SPI transactions. It receives as arguments 4-bit words, that represent the different groups of bits of the 32-bit input shift register (prefix bits, control bits, address bits, data bits and feature bits). The "data" argument is the only exception, since it must be a 16-bit word, although only the first 12 bits are read by the shift register, and the last 4 are ignored. From these arguments this method builds the 32-bit word and sends it on the correct order to the DAC. Before the word is sent to the DAC, the code commands port expander D to set the GPIO that is associated to SYNC (line 460 of Figure 5.13) to the low state '0'. Bits are then sent by the "xfer2()" function.

```
24  >   def __init__(self, spiDevice = spidev.SpiDev(), SYNC = -1, MCP = MCP23S17):···
76
77  >   def writeRegister(self, channel, data):···
100
101 >   def writeAllregisters(self, data):···
122
123 >   def updateChannel(self, channel):···
141
142 >   def updateAllChannels(self):···
155
156 >   def writeAndUpdateChannel(self, channel, data):···
178
179 >   def writeAndUpdateAllChannels(self, data):···
199
200 >   def powerUpChannel(self, channels):···
256
257 >   def powerDownChannel(self, channels, impedance):···
330
331 >   def enableReference(self, mode, flexibleMethod = 0):···
374
375 >   def disableReference(self, mode):···
405
406 >   def reset(self):···
427
428 >   def __setStaticMode(self):···
450
451 >   def __sendWord(self, prefix, control, address, data, feature):···
464
465 >   def __setupGPIO(self):···
474
475 >   def __setupSPI(self):···
```

Figure 5.12. DAC7568 class methods. The most important are the "writeAndUpdateChannel" and "enableReference".

```
451     def __sendWord(self, prefix, control, address, data, feature):
452         first_word = (prefix << 4) | control
453         data1 = (data & 0xF000) >> 12
454         second_word = (address << 4) | data1
455         data2 = (data & 0x0FF0) >> 4
456         third_word = data2
457         data3 = (data & 0x000F)
458         fourth_word = (data3 << 4) | feature
459
460         self.__MCP.digitalWrite(self.__SYNC, self.__MCP.LEVEL_LOW)
461         self.__setupSPI()
462         self.__spi.xfer2([first_word, second_word, third_word, fourth_word])
463         self.__MCP.digitalWrite(self.__SYNC, self.__MCP.LEVEL_HIGH)
```

Figure 5.13. Content of the "__sendWord" private method. This method is responsible for performing all the DAC7568 SPI operations. It receives as arguments the different groups of bits of the DAC7568 32-bit input shift register. All the arguments must be 4-bit words, except for the "data" argument that must be a 16-bit word, where the first 12 bits are the valid sequence and the 4 last bits are ignored.

The “writeAndUpdateChannel()” method performs a write and update operation of the DAC7568 on a specific channel. It takes in as arguments a letter between A and H, that indicates the channel that we wish to write into, and a 12-bit value or an integer number between 0 and 4095. The method configures the bit groups of the 32-bit input shift register of the DAC7568 (prefix, control, address, data and feature bits), and then calls the “__sendWord()” method with these values as arguments. The “data_bits” variable transforms the “data” argument of 12 bits to a 16-bit value in line 173, in order to be valid as argument of the “__sendWord()”. The address of the channel is obtained by a dictionary that the DAC7568 possesses, that makes a correspondence of letters to binary addresses. In the end of the code the “__chData” dictionary attribute is updated.

```
156     def writeAndUpdateChannel(self, channel, data):
157         """
158         Write to selected DAC Input Register and update respective DAC register with the value data
159 
160         Parameters:
161         channel -- letter of the channel of the DAC (A, B, C, D, E, F, G or H)
162         data -- the data to be written and updated on the DAC channel. Must be an integer
163                 less then 4096
164         """
165         assert self.__isInitialized == True
166         assert channel in self.__chData
167         assert type(data) is int
168         assert data < 4096
169 
170         prefix_bits = 0b0000
171         control_bits = 0b0011
172         address_bits = DAC7568.CHANNEL_ADRESSES[channel]
173         data_bits = data << 4
174         feature_bits = 0b0000
175 
176         self.__sendWord(prefix_bits, control_bits, address_bits, data_bits, feature_bits)
177         self.__chData[channel] = data
```

Figure 5.14. Code of the “writeAndUpdateChannel()” method. This method sets the respective bit groups to the correct values in order to perform a write and update channel operation. The method takes in as argument a letter (between A and H) and a value of 12 bits. On line 173 the argument “data” is transformed to a 16-bit value and stored in the variable “data_bits” which is the one that is inserted as argument in the “__sendWord” method.

5.3.3 The MAX1240 class

The ADC’s interface was easier to design, since this device doesn’t have any feature. The only method created performs a reading operation. Of course the “__setupSPI()” and “__setupGPIO()” methods were still created, to fit the design of the project, although they are equal to the ones of the DAC7568 and the MCP23S17. The constructor of the MAX1240 class, Figure 5.15, takes as arguments a SpiDev instance, a pin number for the SHDN pin of MAX[5] (in the case we would want to try to control this pin by software), a MCP23S17 instance and a number corresponding to the GPIO pin of the port expander D, that will be associated to the chip-select of the ADC. As was done in the other classes, the constructor associates the arguments of the instance to attributes and configures the specified port expander’s GPIO as an output in line 32 of Figure 5.15. The constructor also creates two additional attributes, one to define the SPI mode of the ADC which is with CPOL=0 and CPHA=0, and other to store the current value, read by the ADC, in bits.

[5] This argument is optional.

```
13      def __init__(self, spiDevice = spidev.SpiDev(), SHDN = -1, chip_select = -1, MCP = MCP23S17):
14          """
15          Constructor
16          Initializes all attributes
17
18          Keyword arguments:
19          vref -- reference voltage for voltage calculations
20          SHDN -- three-level shutdown input. When using external reference, must be open
21          chip_select -- pin of the MCP23S17 port expander, to be used as chip select
22          """
23
24          self.__SHDN = SHDN
25          self.__chip_select = chip_select
26          self.__MCP = MCP
27          self.__spi = spiDevice
28          self.__spiMode = 0b00
29          self.__voltageBits = 0
30          self.__isInitialized = False
31
32          self.__setupGPIO()
33          self.__isInitialized = True
```

Figure 5.15. MAX1240 class constructor. Arguments are a SpiDev instance ("spiDevice"), a pin number for the SHDN channel ("SHDN"), a pin number for the ADC chip-select ("chip_select") and a MCP23S17 instance ("MCP").

The "readVoltage" method is shown on Figure 5.16. It reads two sequences of 8 bits, sent by the ADC. The sequences are obtained by setting the respective MCP23S17 GPIO to a low state (line 45) and then, after setting the SPI mode, assigning the return value of the "xfer2" function to the 2-element array variable "data" (line 48). The chip-select of the ADC is then set to high state, to close communications. Each element of the array "data" is assigned to one specific variable ("first_byte" and "second_byte") and then some bit-wise operations are made in order to get the 12-bit value of the voltage read by MAX1240.

```
35      def readVoltage(self):
36          """
37          Reads the voltage as a word of 12 bits. Bytes are sent with MSB first, and bits are clocked
38          at the rising edge o SCLK
39          Returns and integer less then 4096
40          """
41          assert self.__isInitialized == True
42          if self.__SHDN != -1:
43              self.__MCP.digitalWrite(self.__SHDN, self.__MCP.LEVEL_HIGH)
44
45          self.__MCP.digitalWrite(self.__chip_select, self.__MCP.LEVEL_LOW)
46          time.sleep(0.01) #wait 9 us for conversion to complete (time specified is actually 7.5 us)
47          self.__setupSPI()
48          data = self.__spi.xfer2([0, 0])
49          self.__MCP.digitalWrite(self.__chip_select, self.__MCP.LEVEL_HIGH)
50
51          first_byte = data[0]
52          second_byte = data[1]
53          # The first bit is '1' and only represents the end of conversion
54          # so it must be deleted
55          first_byte = (first_byte & 0x7F) << 5
56          second_byte = second_byte >> 3
57          self.__voltageBits = first_byte | second_byte
58
59          if self.__SHDN != -1:
60              self.__MCP.digitalWrite(self.__SHDN, self.__MCP.LEVEL_LOW)
61
62          return self.__voltageBits
```

Figure 5.16. Content of the "readVoltage" method. It returns the value read by the ADC in bits.

5.3.4 The HV Remote class

Finally, all the above classes were combined in a HVREMOTE class (Figure 5.17), which is the one that will be used to perform whatever necessary operation on the board. The class has some constant values defined for the port expanders' chip-select pins (Raspberry GPIO pin numeration), DACs' SYNC pins (MCP23S17 pin numeration), ADC chip-select pin (MCP23S17 pin numeration) and for the addresses of each port expander. These were defined in accordance with the schematics of the board (see section 4.4). Each instance of this class will correspond to one board and was defined with four arguments: a SpiDev instance, a RPi.GPIO instance, the board id number and the pin number of the Raspberry Pi that will act as the board chip-select signal. These were associated to attributes, the same way as was made with the other classes. The constructor of this class (Figure 5.18) also defines private attributes that are instances of MCP23S17, six DAC7568 and one MAX1240 classes, which represent the components existent in the HV Remote boards.

```
18  class HVREMOTE(object):
19      # chip select pin - GPIO board numeration
20      # to add more
21      CHIP_SELECT_SPA=11#15
22      CHIP_SELECT_SPB=11#13
23      CHIP_SELECT_SPC=11
24      CHIP_SELECT_SPD=11#12
25      CS_ADC = 1
26      DAC1_SYNC = 3
27      DAC2_SYNC = 4
28      DAC3_SYNC = 5
29      DAC4_SYNC = 6
30      DAC5_SYNC = 7
31      DAC6_SYNC = 8
32
33      # Addresses of the port expanders
34      PORT_EXPANDER_ADD_A = 0b000 #0b000
35      PORT_EXPANDER_ADD_B = 0b001 #0b001
36      PORT_EXPANDER_ADD_C = 0b011 #0b111
37      PORT_EXPANDER_ADD_D = 0b111 #0b011
38
39      def __init__(self, spiDevice=spidev.SpiDev(), iGPIO = GPIO, board_id = 0, chip_select_hvRemote = 40):···
62
63      def enableDisableAllHvChannels(self, onOff):···
77
78      def enableDisableChannel(self, hvChannelNumber, onOff):···
98
99      def readHVChannel(self, hvChannelNumber):···
121
122     def setHV(self, hvChannelNumber, hvOrder):···
```

Figure 5.17. HVREMOTE class constants and methods.

These instances are all initialized with the constant values defined in the beginning of the HVREMOTE class. The DACs and ADC that need a MCP23S17 instance, receive as argument the attribute "__mcpD()", which as the name implies, corresponds to port expander D instance. In the end of the constructor the Raspberry Pi GPIO port associated to the board chip-select signal is configured as an output (line 55).

The HVREMOTE methods are self-explanatory. The "enableDisableAllHvChannels" either enables or disables all the PMTs channels, by setting port expanders A, B and C GPIOs to high ("onOff" = True) or low ("onOff" = False), respectively. It accomplishes this task by using the MCP23S17 "writeGPIO()" method.

```
39      def __init__(self, spiDevice=spidev.SpiDev(), iGPIO = GPIO, board_id = 0, chip_select_hvRemote = 40):
40          self.__spi = spiDevice
41          self.__board_id = board_id
42          self.__chip_select_hvRemote = chip_select_hvRemote
43          self.__GPIO = iGPIO
44
45          #initalization of the port expanders, dacs and adc
46          self.__mcp1 = MCP23S17(self.__spi, self.PORT_EXPANDER_ADD_A, -1, self.CHIP_SELECT_SPA, self.__GPIO)
47          self.__mcp2 = MCP23S17(self.__spi, self.PORT_EXPANDER_ADD_B, -1, self.CHIP_SELECT_SPB, self.__GPIO)
48          self.__mcp3 = MCP23S17(self.__spi, self.PORT_EXPANDER_ADD_C, -1, self.CHIP_SELECT_SPC, self.__GPIO)
49          self.__mcp4 = MCP23S17(self.__spi, self.PORT_EXPANDER_ADD_D, -1, self.CHIP_SELECT_SPD, self.__GPIO)
50          self.__dac1 = DAC7568(self.__spi, self.DAC1_SYNC, self.__mcp4)
51          self.__dac1.enableReference('flexible', 2)
52          self.__dac2 = DAC7568(self.__spi, self.DAC2_SYNC, self.__mcp4)
53          self.__dac3 = DAC7568(self.__spi, self.DAC3_SYNC, self.__mcp4)
54          self.__dac4 = DAC7568(self.__spi, self.DAC4_SYNC, self.__mcp4)
55          self.__dac5 = DAC7568(self.__spi, self.DAC5_SYNC, self.__mcp4)
56          self.__dac6 = DAC7568(self.__spi, self.DAC6_SYNC, self.__mcp4)
57          self.__adc = MAX1240(self.__spi, -1, self.CS_ADC, self.__mcp4)
58          self.__GPIO.setup(self.__chip_select_hvRemote, self.__GPIO.OUT)
```

Figure 5.18. HVREMOTE class constructor. It has a private attribute for each of the main components of the HV Remote boards (MCP, DAC and ADC).

The “enableDisableChannel()” enables or disables an individual PMT channel, using the “digitalWrite()” method of one of the MCP23S17 instances. It takes an integer number between 1 and 48 in the “hvChannelNumber” argument, and a Boolean value in the “onOff” argument.

The “readHVChannel()” method reads the channel specified by the argument “hvChannelNumber” in bits and returns a high voltage equivalent converted value. To obtain a reading, this method will enable one specific multiplexer and select one of its input ports, thus connecting the MUX’s output to the ADC’s AIN channel. Then it uses the MAX1240 “readVoltage()” method to obtain a reading of the channel in bits.

The “setHV()” method adjusts the voltage a the channel, specified by the argument “hvChannelNumber”, to the value specified by “hvOrder”. This method converts the high voltage value to an equivalent value with a maximum of 12 bits (integer less or equal than 4095). The full code of the HVREMOTE class can be seen in annex 8.6.

In order to know which instances these methods must use (for example: “__mcpD” or “__mcpB”? “__dac1” or “__dac5”?) to perform their tasks, they access a map like the one of annex 8.5, that was also written in Python as a dictionary object. This map has an integer number from 1 to 48 as the keyword and a list as the value. This list contains all the information of the respective channel: the port expander and the GPIO port that controls its enable/disable, the DAC that adjusts its voltage and its respective channel (A, B, C, …), the MUX to which its read loop is connected to and its MUX selection word.

5.4 The GUI and the breadboard assemble

After completing the whole software interface, and having the HVREMOTE class working properly, the design of a GUI (Graphical User Interface) was initiated. It will be used to perform the functional tests that must be made to the HV Remote boards, before they are sent to CERN.

The GUI was written in Python 3 as well, using one of its libraries specifically made to GUI design. The library is named PyQt5 and is essentially a set of bindings for Python for the Qt5 C++ module. Python offers a lot of libraries for this kind of applications, but PyQt5 was chosen, due to having a great number of tools and being very used in many industries, thus having lots of support that can be found. This library is based on objects named Widgets and Layouts. The Widgets can be any feature, like: buttons ("QPushButton"), checkboxes ("QCheckBox"), lists ("QComboBox"), etc… The layouts are sections where widgets can be placed, therefore allowing to organize them along the GUI window.

Every PyQt5 application must have one "QApplication()" instance, that may take command line codes as arguments. This object has a method named "exec_()" which must be called in order for the program to run the GUI application. One more line is essential to the code, which is a call to the method "show()" of the class "QMainWindow". This class creates an instance of the window of the GUI, and its method "show()", as the name implies, shows that window when the code is executed. If the methods "exec_()" and "show()" are never called, nothing will appear, although they can be called in the end of the program.

I started the code by creating a SpiDev and RPi.GPIO instances and configuring them properly, Figure 5.19. On the SpiDev I chose the SPI clock speed as 976 kHz (Table 5.1), because it is the higher speed that the Raspberry Pi has available that is less than the maximum clock frequency that the ADC MAX1240 accepts (which is the device with the lowest clock frequency). Also I opened the SpiDev instance in device 0 ("spi.open(0, 0)"), although this is not relevant since the Pi's CE channels aren't going to be used. The GPIO instance was configured to receive the numeration of the GPIO ports of the Raspberry Pi (instead of the numeration of the BCM's GPIOs). After this, an HVREMOTE instance was created, receiving as arguments: the instances that were initialized before, an id for the board and the Pi's GPIO for the board chip-select signal.

```
1   import sys
2   import spidev
3   import RPi.GPIO as GPIO
4   import time
5   from PyQt5.QtGui import *
6   from PyQt5.QtWidgets import *
7   from PyQt5.QtCore import *
8   import pyqtgraph as pg
9   import pyqtgraph.exporters
10  import numpy as np
11  from RPiHVREMOTE.hvRemote import HVREMOTE
12
13  ##### OPENING AND CONFIGURATION OF SPI INTERFACE AND RPI GIPOs #####
14  spi = spidev.SpiDev()
15  spi.open(0, 0)
16  spi.max_speed_hz = 976000
17  spi.no_cs = True
18  GPIO.setwarnings(False)
19  GPIO.setmode(GPIO.BOARD)
20
21  ##### INITIALIZATION OF HVREMOTE BOARDS #####
22  board1 = HVREMOTE(spi, GPIO, 0x00, 40)
```

Figure 5.19. SpiDev, GPIO and HVREMOTE initializations.

The window of the GUI was created with two tabs, one for enabling/disabling the PMTs channels and adjusting their voltages, and another for reading the HV of the PMTs.

On the first tab (Figure 5.20), the enable/disable feature was implemented with one checkbox (highlighted by the orange rectangle) and a label for each channel (highlighted by the blue rectangles), and two buttons that enable/disable all channels (highlighted by the green rectangles). When these buttons are clicked, all checkboxes are checked/unchecked automatically. Below the checkboxes a set of widgets were created to implement the adjusting voltage feature. This comprises a PMT channel selector in the form of a combo box (red rectangle), two labels, a box to insert the desired HV (black rectangle), and a button that sends the command to the board (green rectangle).

On the reading tab (Figure 5.21), a set of real-time plots will be created, that will show the HV of all the PMTs' channels. These plots were not implemented yet, so a very simple functionality that allows to select one channel and then read its voltage was built, to allow the testing of the HVREMOTE class.

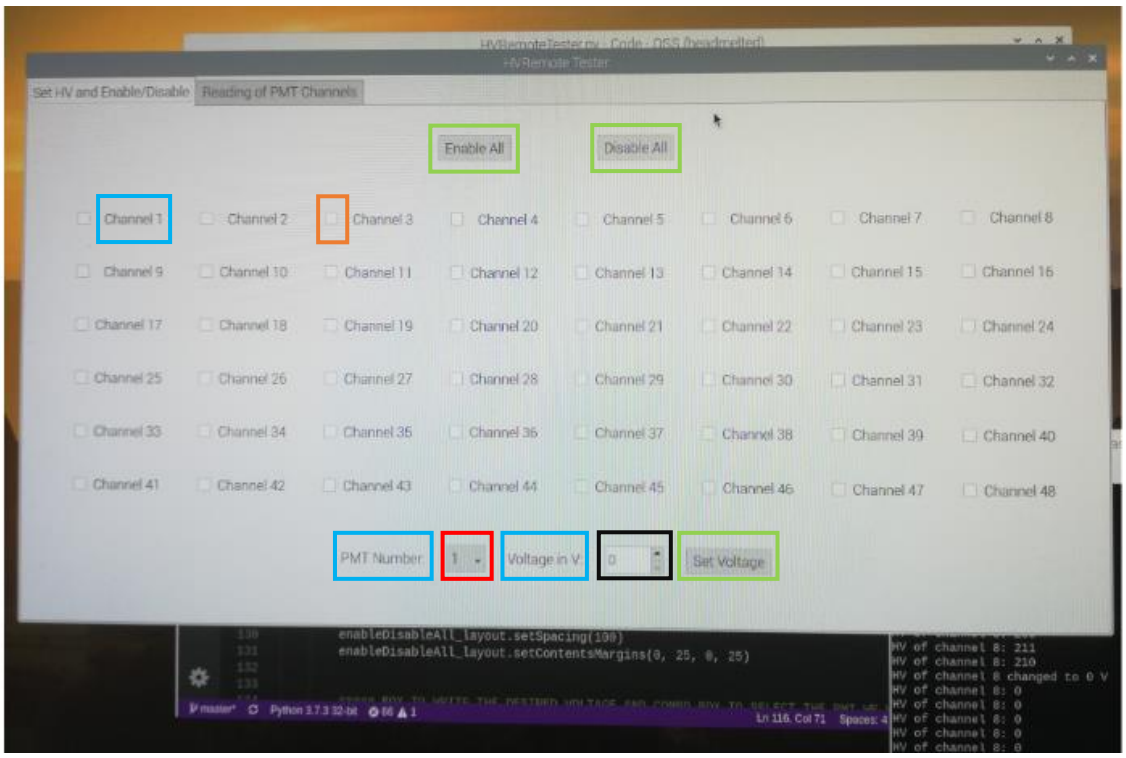

Figure 5.20. Set HV tab of the GUI. Now all channels are disabled, since all checkboxes are unchecked. The orange rectangle signals one checkbox, the blue rectangle signal the labels, the red signals the combo box that allows to select one of the 48 channels, the black rectangle highlights an entry that allows to insert a HV value and the green rectangles highlight the buttons.

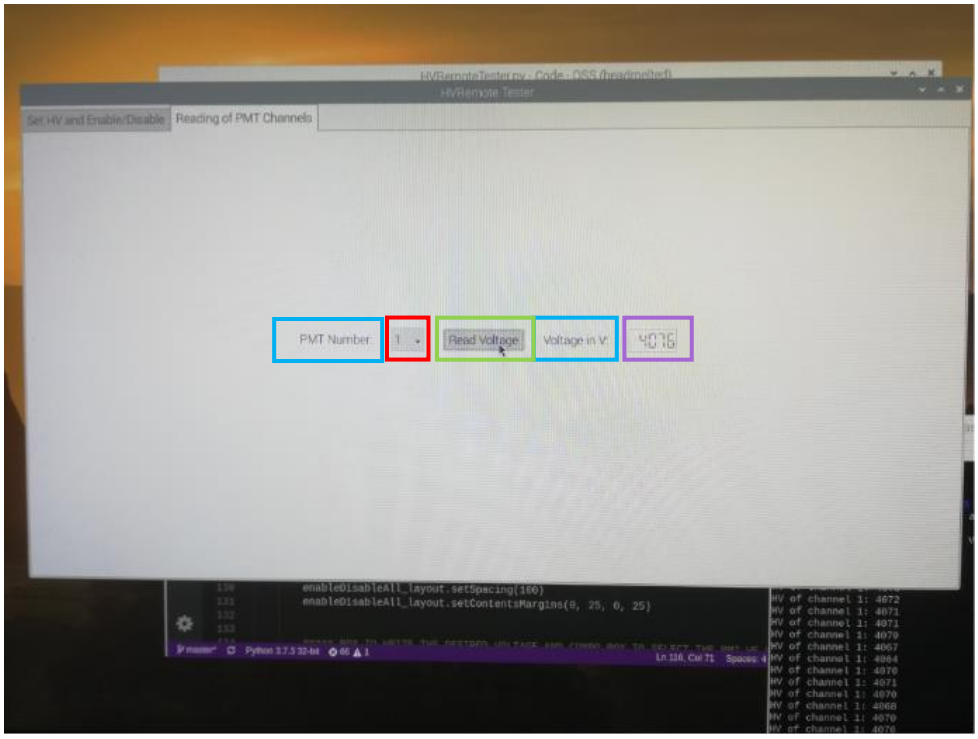

Figure 5.21. The read HV tab. This will be where the real-time plots will be added. For now, only a channel selector (red), a button (green), two labels (blue) and a display (purple) were added to test the HVREMOTE class.

Each label and checkbox respective to each channel, on the set HV tab, corresponds to two widgets ("QLabel" and "QCheckBox") contained in an instance of the class "Channel", shown in Figure 5.22, that inherits the class "QHBoxLayout" (a horizontally divided layout object). It has a counter, "idCounter", that identifies each instance with a number (repective to the PMT numeration). It then creates a checkbox (line 46 of Figure 5.22) and connects the event of changing its state, to the method "change_state" (line 56) that uses the "enableDisableChannel" method of the HVREMOTE class.

```
36  class Channel(QHBoxLayout):
37
38      idCounter = 1
39      def __init__(self, *args, **kwargs):
40          super(Channel, self).__init__(*args, **kwargs)
41
42          self.state = False
43          self.__id = Channel.idCounter
44          Channel.idCounter += 1
45
46          self.__checkBox = QCheckBox()
47          self.__checkBox.setCheckState(Qt.Unchecked)
48          self.__checkBox.stateChanged.connect(self.change_state)
49          self.__channelName = QLabel("Channel " + str(self.__id))
50          self.__channelName.setAlignment(Qt.AlignHCenter | Qt.AlignVCenter)
51
52          self.addWidget(self.__checkBox)
53          self.addWidget(self.__channelName)
54          self.setSpacing(5)
55
56      def change_state(self):···
65
66      def check_box(self):···
71
72      def uncheck_box(self):···
```

Figure 5.22. Code of the "Channel" class. The "change_state" method handles the event of checking/unchecking the checkboxes. The methods "check_box" and "uncheck_box" are used to change the boxes states internally.

The methods “check_box” and “uncheck_box” are used to switch the boxes states internally, when the enable/disable all buttons are clicked. These methods were necessary because, they disconnect the state changing event of the checkboxes, then change their states and finally connect again the state changing event to the “change_state” method.

The rest of the widgets were created in the “MainWindow” class that was already mentioned, which is where the contents of the GUI are created and organized in the application’s window. This class was set to inherit the “QMainWindow” class of the PyQt5 libraries (which is a layout object), therefore allowing the creation of a custom class better suited for the project’s purposes.

The class, with its constructor and other methods is shown in Figure 5.23. To create the tabs of the window, a “QTabWidget” instance is created, with the name “tabs”. To add tabs to this instance we use the method “addTab” on lines 97 and 98 of Figure 5.23. These methods receive as arguments a “QWidget” instance and an optional label. The “QWidget” instances that will represent each tab are initialized in lines 90 and 91 and the respective config methods are called on lines 93 and 94, that create all the content on each tab. An attribute list is also created, where each element corresponds to one instance of the “Channel” class mentioned earlier (line 87 to 89).

The last line of the constructor code (line 100) is the one that indicates which object will be the central widget of the “MainWindow” layout, therefore defining what will appear in the window that will pop up when the code is executed.

```
79  class MainWindow(QMainWindow):
80
81      def __init__(self, *args, **kwargs):
82          super(MainWindow, self).__init__(*args, **kwargs)
83
84          self.setWindowTitle("HVRemote Tester")
85          self.setGeometry(400, 400, 640, 480)
86          self.__chList = []
87          for n in range(1, 49):
88              newChannel = Channel()
89              self.__chList.append(newChannel)
90          self.__setHVTab = QWidget()
91          self.__readTab = QWidget()
92          self.setHVTab_config()
93          self.readTab_config()
94          tabs = QTabWidget()
95          tabs.addTab(self.__setHVTab, "Set HV and Enable/Disable")
96          tabs.addTab(self.__readTab, "Reading of PMT Channels")
97
98          self.setCentralWidget(tabs)
99
100 >    def setHVTab_config(self): ···
160
161 >    def readTab_config(self): ···
190
191 >    def enableAll_clicked(self): ···
197
198 >    def disableAll_clicked(self): ···
204
205 >    def writeButton_clicked(self, chNumber, hvValue): ···
208
209 >    def readButton_clicked(self, chNumber): ···
```

Figure 5.23. Content of the “MainWindow” class. The constructor’s code is presented. The class’ methods definitions are also shown.

The “setHVTab_config” and “readTab_config” build all the widgets that we see on Figure 5.20 and Figure 5.21. The channel’s checkboxes are implemented by creating a “QGridLayout” that is divided in six sections horizontally and eight vertically. Each division of this grid has a 2-time vertically divided layout (created as a “Channel” class instance), where the top section has the checkbox widget and the bottom has a label widget. Another 2-time horizontally divided layout, placed in the top of the application’s window, above the checkboxes, has in each of its sections a button to enable/disable all PMT channels. The “enableAll_clicked” and “disableAll_clicked” are methods that handle the event of clicking on these buttons. They perform a call of the “enableDisableAllHvChannels” method of the HVREMOTE class. In the bottom part of the set HV tab, is the adjust voltage section. This also constitutes a horizontally divided layout that contains a label widget (“QLabel”) followed by a “ComboBox” widget, another “QLabel”, a “QSpinBox” widget and a button. The click event of this button is connected to the “writeButton_clicked” method, that calls the “setHV” method of the HVREMOTE class. All these layouts were then added to a 3-time vertically divided layout, that was set as the main layout of the “__setHVTab” widget. The widgets of the read tab were created in the same way. The full code is shown in annex 8.7.

While the GUI’s code was being written, the HV Remote’s digital components were bought and assembled in a breadboard, with the appropriate connections made (Figure 5.24). These respected the HV Remote’s schematics shown in section 4.4, except for some things that made the wiring in the breadboard easier. For example, the address lines feature of the port expanders was used, instead of using one chip-select for each of them.

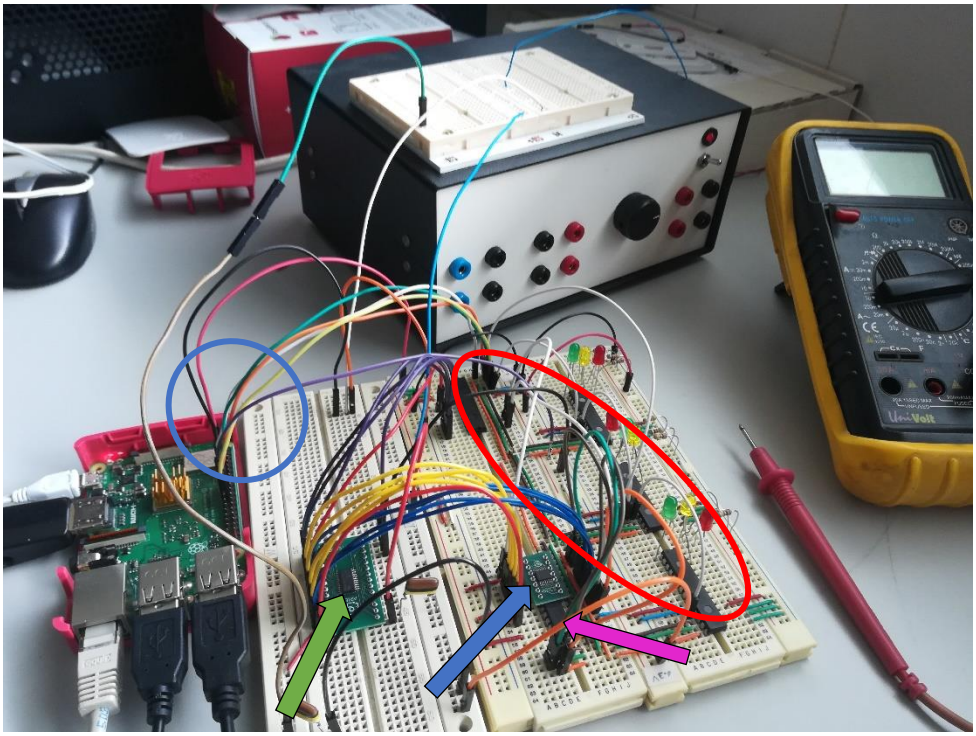

Figure 5.24. HV Remote’s digital control emulated in a breadboard connected to the Raspberry Pi. The cables coming from the Pi (highlighted by the blue circle) are: the SPI MOSI (green), the SPI MISO (orange), the SPI SCLK (yellow), the chip-select of the port expanders (white), the 3.3 V (red and purple) and ground (black). Signalled by the red circle are the port expanders, signalled with the blue arrow is the DAC, signalled by the pink arrow is the ADC and signalled the green is the MUX.

The red and black cables, that are connected to the Pi, on the left of the figure, correspond to the 3.3 V and ground, respectively, that supply the ADC, the DAC and the port expanders. In this setup the other 3.3 V pin of the Raspberry, correspondent to the purple cable, was used to supply the three port expanders on the right side of the breadboard. The MUX is supplied by an external power supply with ±12 V. The green cable is the MOSI line, the orange is the MISO, the yellow is the SCLK and the white is the shared port expanders' chip-select signal. The other cables are just the connections between each component. A voltmeter (right side of Figure 5.24) was used to check connections and the voltages in the component's supply pins. In, total the breadboard contains:

- 4 port expanders, emulating expanders A, B, C and D.
- 1 DAC, emulating DAC1.
- 1 ADC, emulating MAX1240.
- 1 MUX, emulating MUX1.

Some GPIOs of the port expanders were connected to LEDs, in order to test the enable/disable functionality. In Figure 5.26, we can see that the LEDs are blinking, thereby ensuring that the GPIO ports were correctly set to '1'. The GPIO's voltage was also verified with a voltmeter to be certain that the port expanders were giving the 3.3 V. The eight channels of the DAC were connected to the first eight channels of the MUX in order to adjust be able to read all the DAC's channels with the ADC, whose input was connected to the MUX's output. A block diagram of this setup can be seen in Figure 5.25.

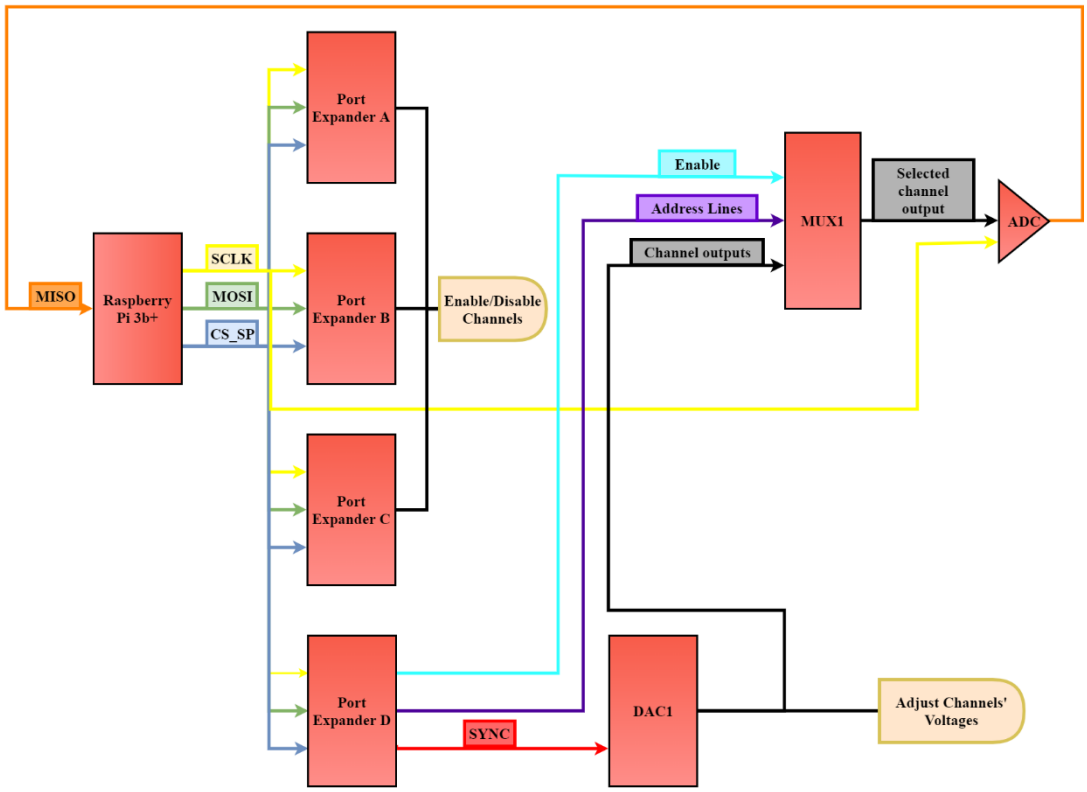

Figure 5.25. Block Diagram of the breadboard setup. The orange line is the MISO connection, the yellow is the SPI clock, the green is the MOSI line, the blue is the port expander's shared chip-select, the purple represents the MUX's address lines, the red is the DAC's SYNC and the black is the DAC's output.

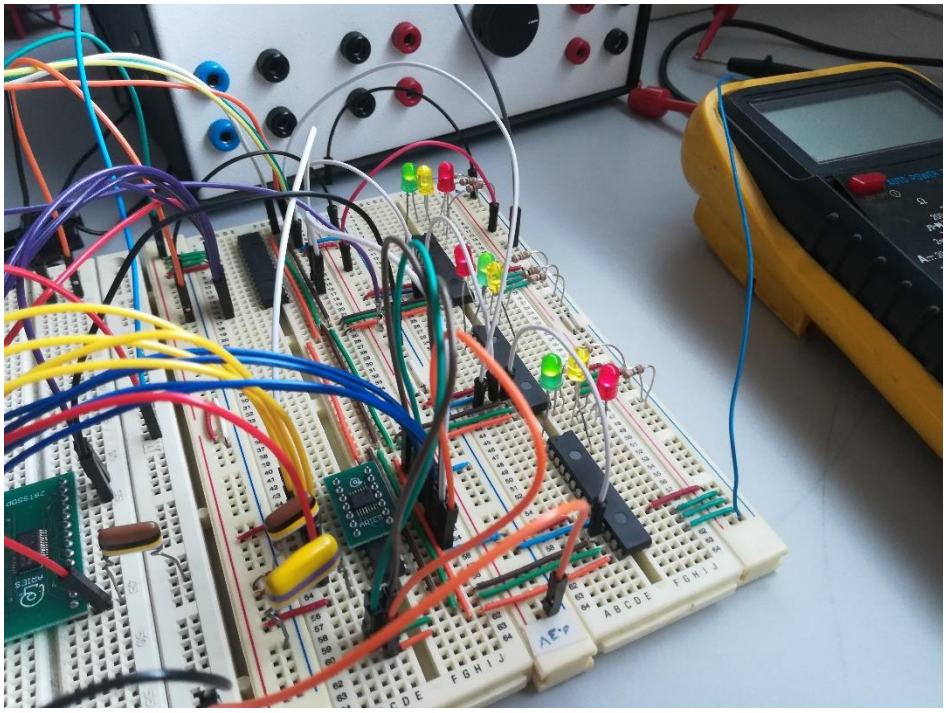

Figure 5.26. Testing the enable/disable channels. We can see that the LEDs are turned on, meaning that at that moment, those channels were enabled.

5.5 Test of the DAC and the of the ADC connected in tandem

A script was written, to return the user's inserted voltage (in bits), when the "Set Voltage" button of the GUI was clicked, and the value read by the ADC, when the "Read Voltage" button in the read tab was clicked. These values were returned to the terminal of the Raspberry, so that they could be compared. An example of these tests is shown in Figure 5.27.

As we can see the ADC's readings usually show values with 20 to 30 counts difference than the one specified. This is equivalent to a difference in voltage between 0.016 V and 0.025 V, which makes up for an error of approximately 0.5% to 0.8 %. The specifications of this project demand an error less than 0.5 %, but the larger error may result from the fact that the recommended capacitors by the datasheets weren't used in the initial tests, since its only purpose was to test the software, and so the stability of the DAC and ADC's voltage reference wasn't guaranteed. In this sense the results were satisfactory, but more rigorous tests will have to be made to evolve to a better system stability and precision.

Even though the results were positive, there was some inconsistencies on the 8th channel of the DAC that showed a difference of 50 counts. This situation is signalled with a red rectangle in Figure 5.26. To further research this issue with more detail, a program (shown in annex 8.8) was written to perform a test across all the valid range of the ADC and DAC, which sport a resolution of 12 bits (so the valid codes go from 0 to 4095). These values written into the DAC (x axis) were then plotted against the readings from the ADC (y axis). This plot can be seen in Figure 5.28.

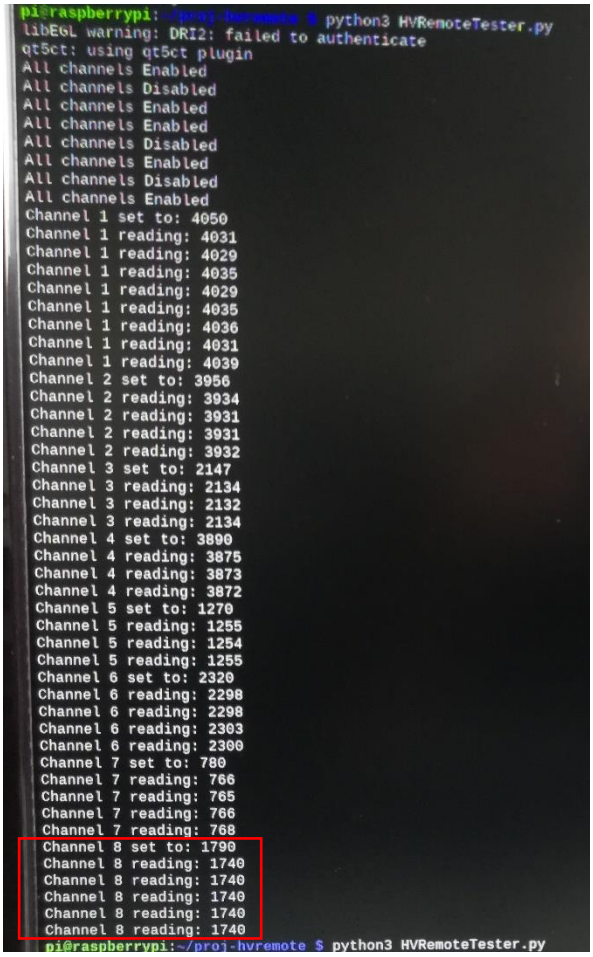

Figure 5.27. Set of write and read tests. It shows the value inserted by the user ("Channel i set to:") and the values read by the ADC ("Channel i reading:"). A difference of 20 to 30 counts can be seen between the value specified to the DAC and the one that is read by the ADC.

Taking a first look to the plot of Figure 5.28, all channels seem to follow the ideal line but with a slight deviation, that accounts to the expected 20 to 30 counts. By zooming in any region of this plot (Figure 5.29) we can see a clear difference between the error of the other channels and the error of channel 8 (black dots).

To see how the error of the channels evolved along all the valid range of data, a plot of the error (value inserted to the DAC minus value read by the ADC) versus the value inserted into the DAC was made. This plot is shown in Figure 5.30. This plot gives us a much better insight as to what the error in this setup really is. For channels 1 to 7 the error is between the 20 and 30 counts. Also, each channel's error seems to vary between 11 and 15 counts along the valid data range. This variation may be due to the missing capacitors in the reference pins of the ADC and DAC. Channel 8 presents a stranger behaviour, having its value rising to around 60 counts and varying in the interval from 60 to 80 counts.

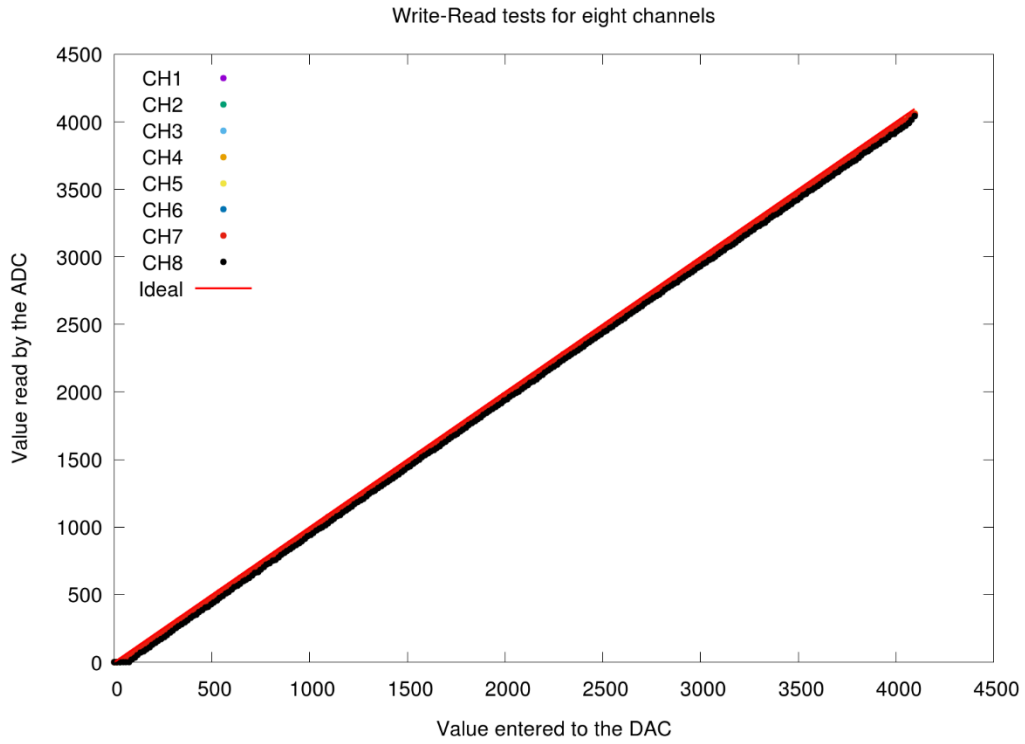

Figure 5.28. Write-Read test for all the DAC's eight channels. This test was made with a step of 15 counts and spans the entire valid range of data (from 0 to 4095). This test was made with a SPI clock frequency of 976 kHz.

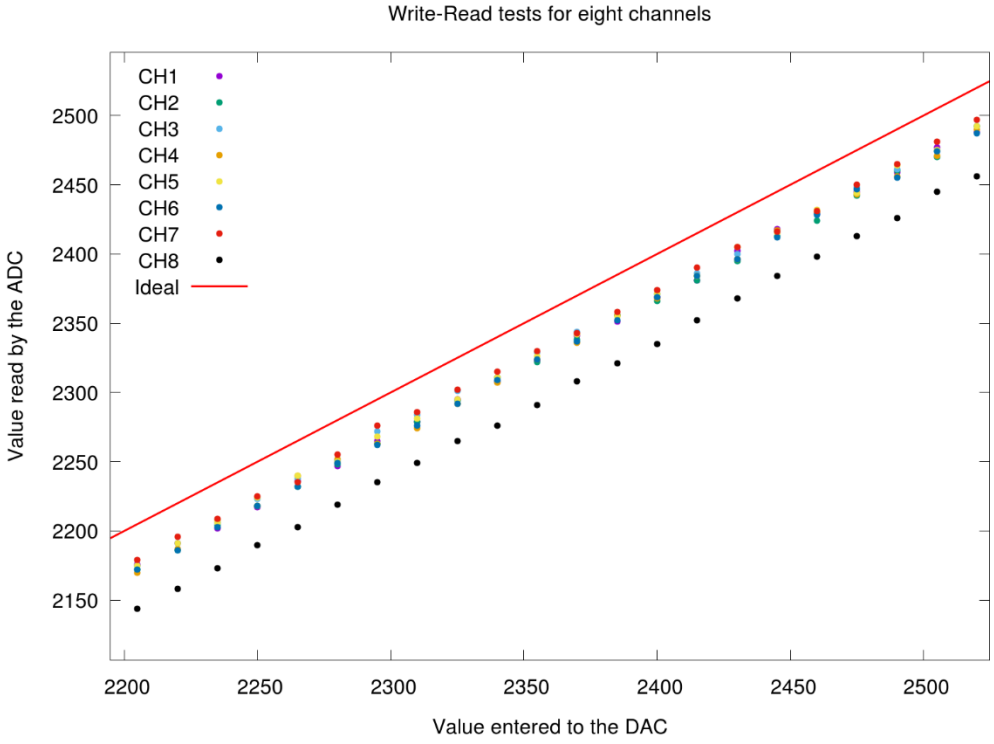

Figure 5.29. Zoom of the middle region of the plot of Figure 5.28.

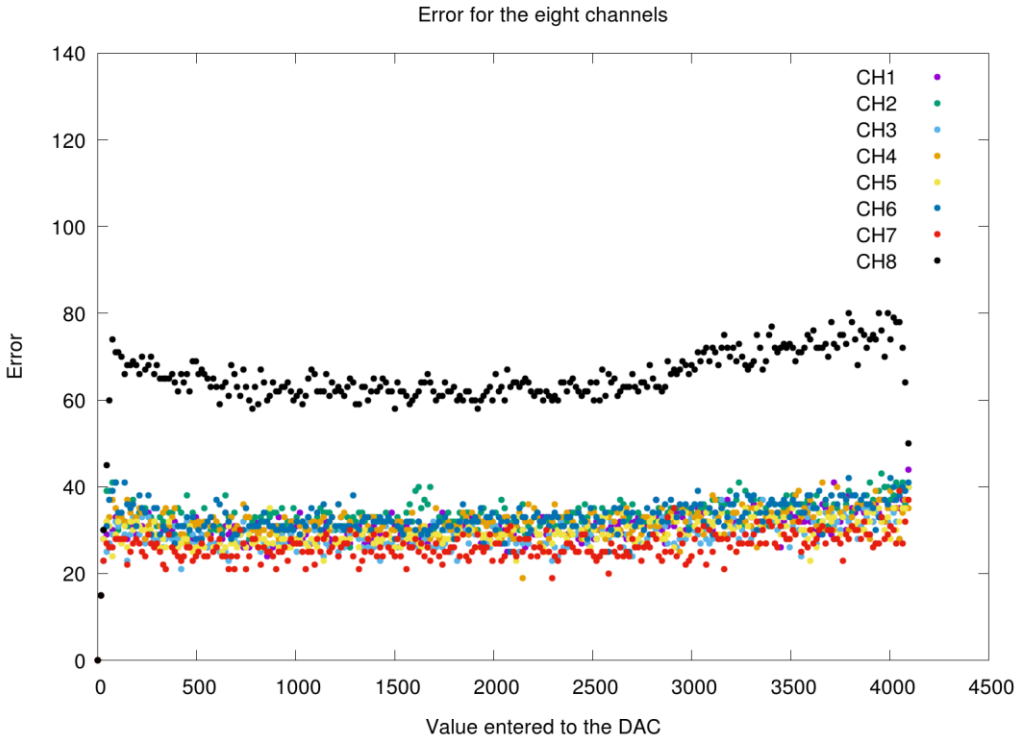

Figure 5.30. Plot of the error of the ADC readings versus value inserted to the DAC. This test was made with the same steps and same clock frequency as the ones above.

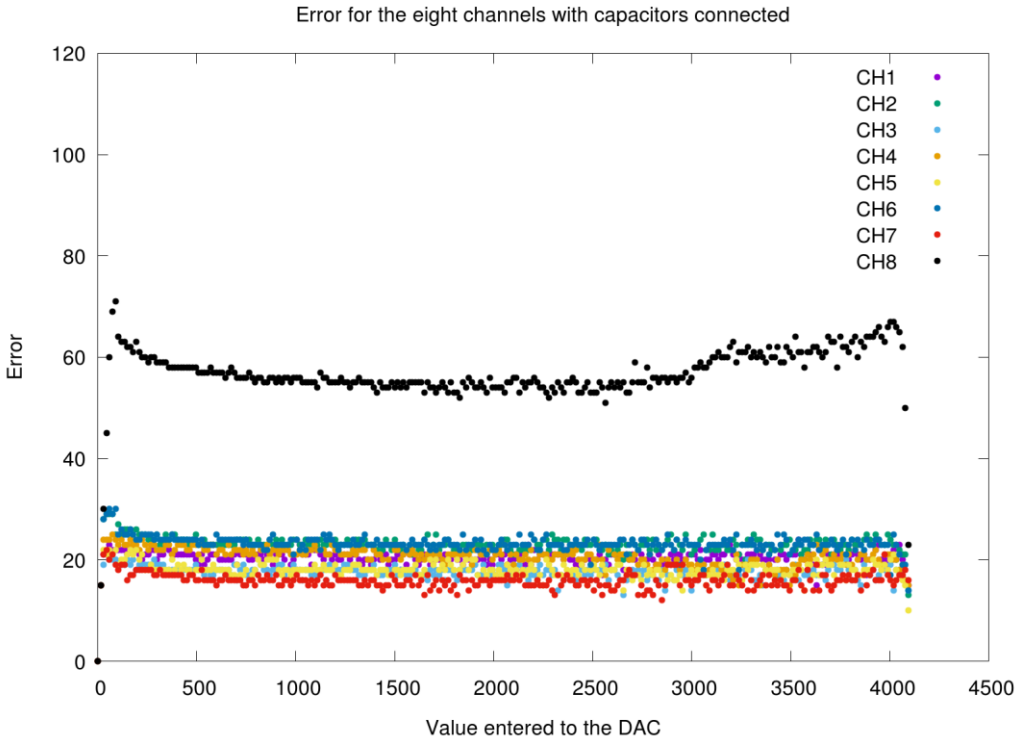

Figure 5.31. Plot of the error of the ADC readings versus value inserted to the DAC, with capacitors connected. This test was made with the same steps and same clock frequency as the ones above.

The same tests were then made with the capacitors connected to see if the error stabilized and to see if it diminished. The results are shown in Figure 5.31. We can see that the error stability in channels 1 to 7, improved significantly. The error has also reduced slightly being now confined to the limits of 16 to 23 counts and having variations between 3 and 5 counts. The behaviour of channel 8 hasn't changed much, although its error reduced to around 57 counts. Table 5.2 summarizes the mean error and the mean variations in each channel, before and after using the capacitors.

Table 5.2. Summary of the mean error and mean error variation for a setup without capacitors and with capacitors.

Channel	Without capacitors		With capacitors	
	Mean error	Mean error variation	Mean error	Mean error variation
1	31	13	20	3
2	33	14	23	4
3	29	13	18	3
4	32	13	21	5
5	29	11	18	3
6	33	15	23	5
7	26	11	16	3
8[6]	66	25	57	12

To track down the source of the problem seen on channel 8, another run was made with the DAC's channels directly connected to the ADC, therefore disconnecting the MUX from the setup. The results are shown in Figure 5.32, and they show that the different error that channel 8 had, disappeared, presenting now a similar error to the other channels, although in the limits of the valid data range, it shows an abrupt increase of the error. This test shows that the MUX has some influence on the values of channel 8. This problem will need to be better studied in the future, by trying to revise all connections, and trying other components, in order to conclude if it's a hardware problem or not.

After these studies, some tests to the SPI clock frequency were made, to see which was the maximum frequency to which the system could communicate properly. Two plots were made (Figure 5.33 and Figure 5.34), one showing the results for the clock frequencies 15.2, 61, 244 and 976 kHz, and another for the frequencies 1.953, 3.1, 7.8 and 15.6 MHz. The first plot crosses the lower frequencies below the expected limit, which is 2.1 MHz and the second goes above this value.

From these plots we can conclude that the error of the ADC's readings is independent from the clock frequency (for frequencies below 2.1 MHz). The lower frequencies, Figure 5.33, show positive results, having all ADC measures a good proximity with the values specified to the DAC, with an error similar to the one discusses above. From the plot of the higher frequencies, Figure 5.34, we can immediately deduce that the 15.6 MHz frequency is not adequate for this project. After removing this speed from the plot (right side of Figure 5.34), we also see that the 7.8 MHz frequency does not fit for this project.

[6] The points at the beginning and at the end of the data range weren't taken in for the calculations of the mean error and mean error variation of channel 8.

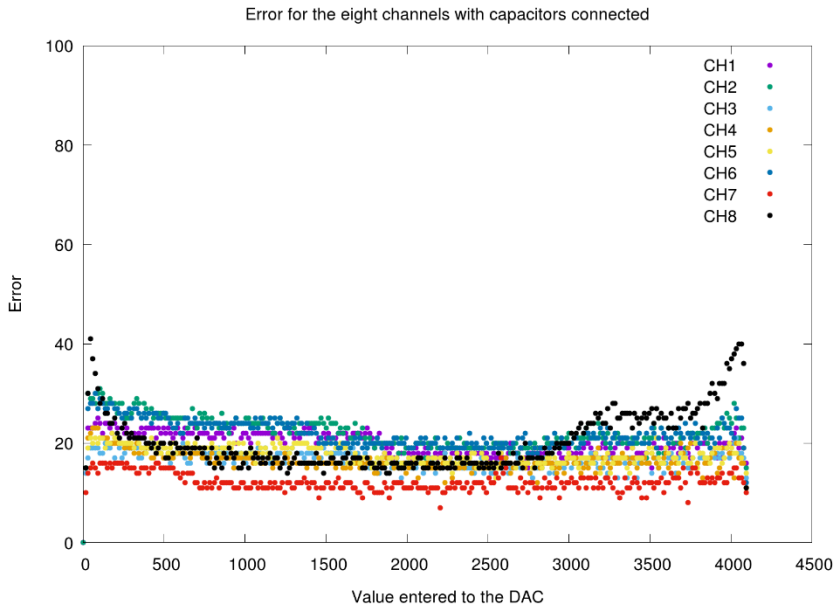

Figure 5.32. Error of the ADC's readings with capacitors connected and without the MUX. The DAC's channels were directly connected to the ADC.

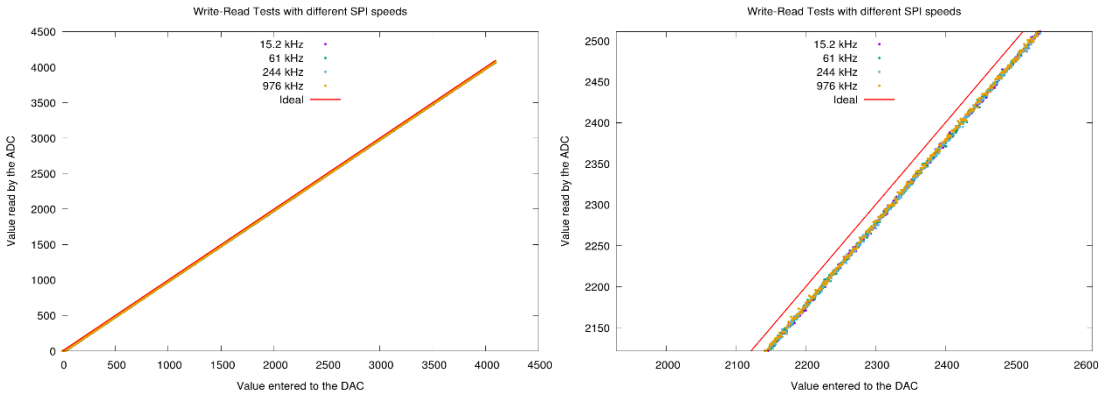

Figure 5.33. SPI clock frequency tests. The plot of the right side is a zoom of the middle region of the left plot. These tests were made with a step of 1 count for the clock frequencies of 15.2, 61, 244 and 976 kHz.

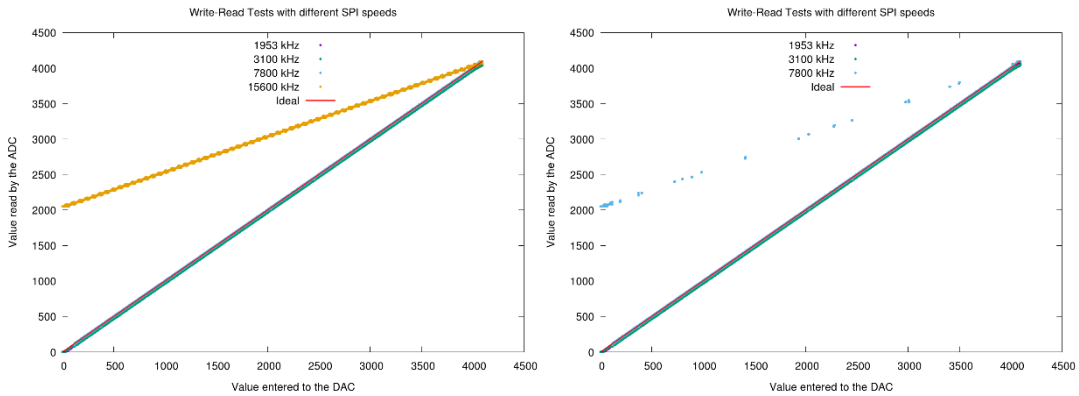

Figure 5.34. SPI clock frequency tests. The plot of the left side shows the tests results for the clock frequencies of 1.953, 3.1, 7.8 and 15.6 MHz, and the plot of the right side shows the same results but without the 15.6 MHz data.

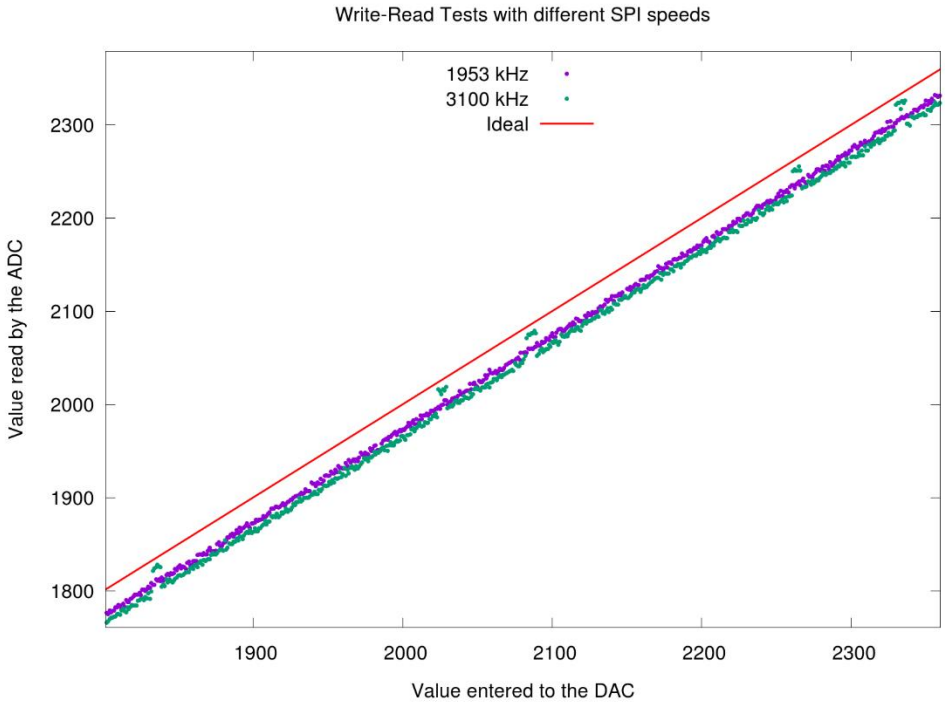

Figure 5.35. SPI clock frequency tests for 3.1 MHz and 1.953 MHz. This plot is a zoom of the middle region of the original plot, that shows the distribution of points with more detail.

Now we are left with two possible clock high frequencies, the 3.1 MHz and 1.953 MHz. A zoom of the plot of the tests to these two frequencies can be seen in Figure 5.35. Both show the same results that were seen in the lower frequencies tests, which tells us that both are valid frequencies for controlling the HV Remote boards, although the 3.1 MHz frequency is not much stable, showing some inconsistencies on the ADC's readings (even though these are closer to the ideal value). This subject will have to be better tested in the HV Remote boards, since if the 3.1 MHz frequency does not show concerning issues on the board's operations and on the PMT channel's HV stability, it may be used for the project.

The longest bit transaction that occurs in the board will be when we adjust the PMT's voltages. To do this, the board has to send 16 bits to port expander D, to command it to set one DAC's SYNC signal high, and then will need to send 32 bits to this DAC to adjust one of its channel's voltage. In total this operation will require 48 bits, which are transferred in 48 clock cycles. If the board uses a clock with 3.1 MHz, this operation will take, in total, approximately 15.5 µs, while if a clock with 1.953 MHz is used, the operation would take 24.6 µs.

6 Conclusion

6.1 Summary of the activities and goals achieved

The main part of this work, which consisted on writing a SPI interface for each of the HV Remote components, and for the board itself, was concluded. The Raspberry Pi was also tested, and for now it seems that it can handle the communication with all the boards.

Some tests were performed to each component, isolated from the others, in order to see if all the functions were working properly. These singular tests were made by connecting the components on a breadboard and setting the correct connections. With the help of a voltmeter the voltages in each channel were constantly measured in order to see if they matched the expected values. By doing this, it was much easier to track errors in the code and to understand how to properly connect the devices to the Raspberry Pi.

There were some difficulties at the beginning regarding how the code should be written for the classes to complement each other, when writing the code for the HVREMOTE class, so some changes were made to the other classes. In the end, all the HV Remote's functions were properly implemented in software, although some inconsistencies between the specified values to the DAC and the values read by the ADC were seen in Figure 5.27, that account to an error around 20 counts. After using the algorithm of annex 8.8, to further investigate this issue, a strange phenomenon was observed on channel 8 and the suspicions of an error on the voltage readings were confirmed.

The development of the GUI, which is what will be used to perform the boards' tests, has been started, with the adjust voltage and enable/disable features already completed and tested with the components assembled in the breadboard. As was seen in **Erro! A origem da referência não foi encontrada.** and **Erro! A origem da referência não foi encontrada.** these tests showed good results in terms of software.

As was seen in Figure 5.26 and Figure 5.27 the correct execution of the battery of tests validates the software we have developed in object oriented Python to manage the SPI communications between the Raspberry Pi and the prototype of the SPI-controller hardware, that will be in the HV Remote boards.

6.2 Future work

In the HVREMOTE class, a code to read the temperature probes, must still be developed. This was not accomplished, since breadboard friendly versions of these components weren't available, and so the code could not be tested in order to see if things were working properly. When the boards are finished, with the temperature sensors embedded, a method for this function must be created in the HVREMOTE class.

The errors that were observed in some readings of the DAC channels must be better understood, since it is of utmost importance that the DAC's voltages have a good precision and stability. So, more rigorous tests must be made, and probably different setups, a different ADC or even another ADC of the same model must be used. From the results of Figure 5.32, we are induced that the MUX may also have an influence on the measures that were obtained, so a different MUX or another of the same model must be tried as well.

In the end, the errors may also be caused due to the fact that this experiment was made in a breadboard with a lot of wires making the connections, hence the importance of doing these tests on the boards. Another important thing to note, is that the Raspberry Pi was supplying all the components with 3.3 V, so there is also the possibility that the Pi couldn't supply enough current to all the components, which may result in variations in the reference pins of the DAC and ADC.

Although part of the GUI has been developed it is the part that needs substantial improvements. In the read tab of the GUI's window, the real-time plots still need to be added in order to monitor the precision and stability of the PMTs' HV and of the board's temperature. More features for the GUI may also be added if needed, when the definitive tests are being specified and the test flow is defined.

In the future, tests at different temperature regimes must also be made, in order to see how the board and its components behave in harsher conditions. The results of these may also be useful to understand how the boards should be positioned in the crates, in order to have a better air flow for removing the heat developed while in operation.

6.3 Personal reflection

Overall this work was a great experience since it was an opportunity to learn how to use a small-scale processor for control and test purposes (in this case it was a Raspberry Pi), which is a technology that is used in lots of applications. This work also provided experience on programming with Python and some of its modules for the first time, namely the SpiDev and the PyQt5 modules.

The development of a communication network connecting with several chips and based in the SPI protocol was also very educational, since previously I only had a general idea of how these digital communication interfaces operated. I believe that now I feel more comfortable dealing with challenges related to digital chips and serial interfaces, due to this experience.

7 References

[1] [Online] “https://home.cern/science/accelerators/large-hadron-collider”.

[2] ATLAS collaboration, “ATLAS detector and physics performance: Technical Design Report”, CERN/LHCC 99-15.

[3] ATLAS collaboration, “The ATLAS Experiment at the CERN Large Hadron Collider”, 2008.

[4] ATLAS collaboration, “Tile Calorimeter Technical Design Report”, CERN/LHCC/96-42.

[5] ATLAS collaboration, “The ATLAS Tile Calorimeter”, A. Henriques, 2015.

[6] ATLAS collaboration, “The optical instrumentation of the ATLAS Tile Calorimeter”, 2013.JINST 8 P01005.

[7] D. Calver, “The High Voltage distribution system of the ATLAS Tile Calorimeter and its performance during data taking”, 2018 JINST 13 P08006.

[8] ATLAS Collaboration, “ATLAS high-level trigger, data-acquisition and controls: Technical Design Report”, CERN-LHCC-2003-022, CERN, Geneva, 2003.

[9] [Online] “https://home.cern/news/news/accelerators/record-luminosity-well-done-lhc”.

[10] [Online] “https://home.cern/science/accelerators/high-luminosity-lhc”.

[11] P. Vankov, “ATLAS upgrade for the HL-LHC: meeting the challenges of a five-fold increase in collision rate”, Hamburg, Germany.

[12] [Online] “https://home.cern/news/news/experiments/new-small-wheels-set-atlas-track-high-luminosity”.

[13] Eduardo Valdes Santurio, “Upgrade of Tile Calorimeter of the ATLAS Detector for the High Luminosity LHC”, Stockholm, Sweden, 2017.

[14] Atlas-tile-demonstrator, ATLAS TileCal demonstrator project, CERN Demonstrator webpage.

[15] F. Vazeille, “Performance of a Remote High Voltage Power Supply 2 for the Phase II Upgrade of the ATLAS Tile Calorimeter”, CERN, Geneva, 2015.

[16] “Tibbo Programmable Hardware Manual”, Tibbo Technology, 2015.

[17] “16-Bit I/O Expander with Serial Interface”, Microchip Technology Inc., 2007.

[18] “12-/14-16-Bit, Octal Channel, Ultralow Glitch, Voltage Output Digital-to-Analog Converters with 2.5, 2ppm/ºC Internal Reference,” Texas Instruments, 2009, Revised 2014.

[19] “Serial Analog-To-Digital Converters with Autopower Down,” Texas Instruments Incorporated, 2000, Revised 2010.

[20] “+2.7 V, Low-Power, 12-bit Serial ADCs in 8-pin SO” Maxim Integrated Products, 2010.

[21] “8-bit FET Bus Switch 2.5-V/3.3-V Low-Voltage High-Bandwidth Bus Switch” Texas Instruments, 2003, Revised 2005.

[22] “MUX36xxx 36-V, Low-Capacitance, Low-Leakage-Current, Precision Analog Multiplexers”, Texas Instruments, 2016, Revised 2018.

[23] “Low-Cost, Current Output Temperature Transducer”, Analog Devices Inc., Norwood, USA, 2003.

[24] “2 - Terminal IC 1.2 V Reference,” Analog Devices, Norwood, USA, 2004.

[25] “Single-Ended 16-Channel/Differential 8-Channel CMOS Analog Multiplexers” Burr-Brown Products from Texas Instruments, 1988, Revised 2003.

[26] “8-Channel Level Translators,” Maxim Integrated, 2008.

[27] F. Martins, L. Gurriana, L. Seabra, G. Evans, A. Maio, C. Rato, J. Sabino, J. Soares Augusto, “Control System for ATLAS TileCal HVRemote boards”.

[28] [Online] “https://www.raspberrypi.org/”.

[29] Broadcom, “BCM2835 ARM Peripherals”, Broadcom Corporation, 2012.

[30] [Online] “http://www.gammon.com.au/spi”.

[31] [Online]“https://github.com/petrockblog/RPiMCP23S17/blob/master/RPiMCP23S17/MCP23S17.py”.

8 Annex

8.1 Tables of all the DAC7568 commands

DB31	DB30-DB28	DB27	DB26	DB25	DB24	DB23	DB22	DB21	DB20	DB19	DB18	DB17	DB16-DB10	DB9	DB8	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	DESCRIPTION
0	Don't Care	C3	C2	C1	C0	A3	A2	A1	A0	D16	D15	D14	D13-D7	D6	D5	D4	D3	D2	D1	F3	F2	F1	F0	GENERAL DATA FORMAT FOR 16-BIT DAC8568
0	Don't Care	C3	C2	C1	C0	A3	A2	A1	A0	D14	D13	D12	D11-D5	D4	D3	D2	D1	X	X	F3	F2	F1	F0	GENERAL DATA FORMAT FOR 14-BIT DAC8168
0	Don't Care	C3	C2	C1	C0	A3	A2	A1	A0	D12	D11	D10	D9-D3	D2	D1	X	X	X	X	F3	F2	F1	F0	GENERAL DATA FORMAT FOR 12-BIT DAC7568
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Reserved Bit - Not valid; device does not perform to specified conditions
Write to Selected DAC Input Register																								
0	X	0	0	0	0	0	0	0	0	Data										X	X	X	X	Write to input register - DAC Channel A
0	X	0	0	0	0	0	0	0	1		Data									X	X	X	X	Write to input register - DAC Channel B
0	X	0	0	0	0	0	0	1	0			Data								X	X	X	X	Write to input register - DAC Channel C
0	X	0	0	0	0	0	0	1	1				Data							X	X	X	X	Write to input register - DAC Channel D
0	X	0	0	0	0	0	1	0	0				Data							X	X	X	X	Write to input register - DAC Channel E
0	X	0	0	0	0	0	1	0	1				Data							X	X	X	X	Write to input register - DAC Channel F
0	X	0	0	0	0	0	1	1	0				Data							X	X	X	X	Write to input register - DAC Channel G
0	X	0	0	0	0	0	1	1	1				Data							X	X	X	X	Write to input register - DAC Channel H
0	X	0	0	0	0	1	X	X	X				X							X	X	X	X	Invalid code - No DAC channel is updated
0	X	0	0	0	0	1	1	1	1				Data							X	X	X	X	Broadcast mode - Write to all DAC channels
Update Selected DAC Registers																								
0	X	0	0	0	1	0	0	0	0				Data							X	X	X	X	Update DAC register - DAC Channel A
0	X	0	0	0	1	0	0	0	1				Data							X	X	X	X	Update DAC register - DAC Channel B
0	X	0	0	0	1	0	0	1	0				Data							X	X	X	X	Update DAC register - DAC Channel C
0	X	0	0	0	1	0	0	1	1				Data							X	X	X	X	Update DAC register - DAC Channel D
0	X	0	0	0	1	0	1	0	0				Data							X	X	X	X	Update DAC register - DAC Channel E
0	X	0	0	0	1	0	1	0	1				Data							X	X	X	X	Update DAC register - DAC Channel F
0	X	0	0	0	1	0	1	1	0				Data							X	X	X	X	Update DAC register - DAC Channel G
0	X	0	0	0	1	0	1	1	1				Data							X	X	X	X	Update DAC register - DAC Channel H
0	X	0	0	0	1	1	X	X	X				X							X	X	X	X	Invalid code - No DAC channel is updated
0	X	0	0	0	1	1	1	1	1				Data							X	X	X	X	Broadcast mode - Update all DAC registers

DB31	DB30-DB28	DB27	DB26	DB25	DB24	DB23	DB22	DB21	DB20	DB19	DB18	DB17	DB16-DB10	DB9	DB8	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	DESCRIPTION
0	Don't Care	C3	C2	C1	C0	A3	A2	A1	A0	D16	D15	D14	D13-D7	D6	D5	D4	D3	D2	D1	F3	F2	F1	F0	GENERAL DATA FORMAT FOR 16-BIT DAC8568
0	Don't Care	C3	C2	C1	C0	A3	A2	A1	A0	D14	D13	D12	D11-D5	D4	D3	D2	D1	X	X	F3	F2	F1	F0	GENERAL DATA FORMAT FOR 14-BIT DAC8168
0	Don't Care	C3	C2	C1	C0	A3	A2	A1	A0	D12	D11	D10	D9-D3	D2	D1	X	X	X	X	F3	F2	F1	F0	GENERAL DATA FORMAT FOR 12-BIT DAC7568
Write to Selected DAC Input Register and Update All DAC Registers																								
0	X	0	0	1	0	0	0	0	0	Data										X	X	X	X	Write to DAC input register Ch A and update all DAC registers (SW LDAC)
0	X	0	0	1	0	0	0	0	1	Data										X	X	X	X	Write to DAC Input Register Ch B and update all DAC registers (SW LDAC)
0	X	0	0	1	0	0	0	1	0	Data										X	X	X	X	Write to DAC Input Register Ch C and update all DAC registers (SW LDAC)
0	X	0	0	1	0	0	0	1	1	Data										X	X	X	X	Write to DAC Input Register Ch D and update all DAC registers (SW LDAC)
0	X	0	0	1	0	0	1	0	0	Data										X	X	X	X	Write to DAC Input Register Ch E and update all DAC registers (SW LDAC)
0	X	0	0	1	0	0	1	0	1	Data										X	X	X	X	Write to DAC Input Register Ch F and update all DAC registers (SW LDAC)
0	X	0	0	1	0	0	1	1	0	Data										X	X	X	X	Write to DAC Input Register Ch G and update all DAC registers (SW LDAC)
0	X	0	0	1	0	0	1	1	1	Data										X	X	X	X	Write to DAC Input Register Ch H and update all DAC registers (SW LDAC)
0	X	0	0	1	0	1	X	X	X	X										X	X	X	X	Invalid code - No DAC Channel is updated
0	X	0	0	1	0	1	1	1	1	Data										X	X	X	X	Broadcast mode - Write to all DAC input registers and update all DAC registers (SW LDAC)
Write to Selected DAC Input Register and Update Respective DAC Register																								
0	X	0	0	1	1	0	0	0	0	Data										X	X	X	X	Write to DAC input register Ch A and update DAC register Ch A
0	X	0	0	1	1	0	0	0	1	Data										X	X	X	X	Write to DAC Input Register Ch B and update DAC register Ch B
0	X	0	0	1	1	0	0	1	0	Data										X	X	X	X	Write to DAC Input Register Ch C and update DAC register Ch C
0	X	0	0	1	1	0	0	1	1	Data										X	X	X	X	Write to DAC Input Register Ch D and update DAC register Ch D
0	X	0	0	1	1	0	1	0	0	Data										X	X	X	X	Write to DAC Input Register Ch E and update DAC register Ch E
0	X	0	0	1	1	0	1	0	1	Data										X	X	X	X	Write to DAC Input Register Ch F and update DAC register Ch F
0	X	0	0	1	1	0	1	1	0	Data										X	X	X	X	Write to DAC Input Register Ch G and update DAC register Ch G
0	X	0	0	1	1	0	1	1	1	Data										X	X	X	X	Write to DAC Input Register Ch H and update DAC register Ch H
0	X	0	0	1	1	1	X	X	X	X										X	X	X	X	Invalid code - No DAC channel is updated
0	X	0	0	1	1	1	1	1	1	Data										X	X	X	X	Broadcast mode - Write to all DAC input registers and update all DAC registers (SW LDAC)

DB31	DB30-DB28	DB27	DB26	DB25	DB24	DB23	DB22	DB21	DB20	DB19	DB18	DB17	DB16-DB10	DB9	DB8	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	DESCRIPTION
0	Don't Care	C3	C2	C1	C0	A3	A2	A1	A0	D16	D15	D14	D13-D7	D6	D5	D4	D3	D2	D1	F3	F2	F1	F0	GENERAL DATA FORMAT FOR 16-BIT DAC8568
0	Don't Care	C3	C2	C1	C0	A3	A2	A1	A0	D14	D13	D12	D11-D5	D4	D3	D2	D1	X	X	F3	F2	F1	F0	GENERAL DATA FORMAT FOR 14-BIT DAC8168
0	Don't Care	C3	C2	C1	C0	A3	A2	A1	A0	D12	D11	D10	D9-D3	D2	D1	X	X	X	X	F3	F2	F1	F0	GENERAL DATA FORMAT FOR 12-BIT DAC7568
Power-Down Commands																								
0	X	0	1	0	0	X	X	X	X	X	X	X	X	0	0	DAC H	DAC G	DAC F	DAC E	DAC D	DAC C	DAC B	DAC A	Power-up DAC A, B, C, D, E, F, G, H by setting respective bit to '1'
0	X	0	1	0	0	X	X	X	X	X	X	X	X	0	1	DAC H	DAC G	DAC F	DAC E	DAC D	DAC C	DAC B	DAC A	Power-down DAC A, B, C, D, E, F, G, H, 1kΩ to GND by setting respective bit to '1'
0	X	0	1	0	0	X	X	X	X	X	X	X	X	1	0	DAC H	DAC G	DAC F	DAC E	DAC D	DAC C	DAC B	DAC A	Power-down DAC A, B, C, D, E, F, G, H, 100kΩ to GND by setting respective bit to '1'
0	X	0	1	0	0	X	X	X	X	X	X	X	X	1	1	DAC H	DAC G	DAC F	DAC E	DAC D	DAC C	DAC B	DAC A	Power-down DAC A, B, C, D, E, F, G, H, High-Z to GND by setting respective bit to '1'
Internal Reference Commands																								
0	X	1	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0	Power down internal reference - static mode (default), must use external reference to operate device; see Table 3
0	X	1	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1	Power up internal reference - static mode; see Table 2 (NOTE: When all DACs power down, the reference powers down; when any DAC powers up, the reference powers up)
0	X	1	0	0	1	X	X	X	X	1	0	0	X	X	X	X	X	X	X	X	X	X	X	Power up internal reference - flexible mode; see Table 4 (NOTE: When all DACs power down, the reference powers down; when any DAC powers up, the reference powers up)
0	X	1	0	0	1	X	X	X	X	1	0	1	X	X	X	X	X	X	X	X	X	X	X	Power up internal reference all the time regardless of state of DACs - flexible mode; see Table 5
0	X	1	0	0	1	X	X	X	X	1	1	0	X	X	X	X	X	X	X	X	X	X	X	Power down internal reference all the time regardless of state of DACs - flexible mode; see Table 6 (NOTE: External reference must be used to operate device)
0	X	1	0	0	1	X	X	X	X	0	0	0	X	X	X	X	X	X	X	X	X	X	X	Switching internal reference mode from flexible mode to static mode
Reserved Bits																								
0	X	1	0	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Reserved Bit - not valid; device does not perform to specified conditions
0	X	1	0	1	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Reserved Bit - not valid; device does not perform to specified conditions
0	X	1	1	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Reserved Bit - not valid; device does not perform to specified conditions
0	X	1	1	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Reserved Bit - not valid; device does not perform to specified conditions
0	X	1	1	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Reserved Bit - not valid; device does not perform to specified conditions
0	X	1	1	1	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Reserved Bit - not valid; device does not perform to specified conditions

8.2 Port Expander's Connections

Note: Only Expander A and D are shown. Expander B and C have the same kind of connections as expander A, only differing in the channels' numeration.

8.2.1 Port Expander A

Physical Pin	Pin Name	Connected Signal
1	GPB0	CH1_EN3
2	GPB1	CH2_EN3
3	GPB2	CH3_EN3
4	GPB3	CH4_EN3
5	GPB4	CH1_EN4
6	GPB5	CH2_EN4
7	GPB6	CH3_EN4
8	GPB7	CH4_EN4
9	V_{DD}	V3
10	V_{SS}	DGND
11	$\overline{CS}$	CS_SPA
12	SCK	SCLK
13	SI	DIN
14	SO	DOUTA
15	A0	DGND
16	A1	DGND
17	A2	DGND
18	$\overline{RESET}$	V3
19	INTA	Not Connected
20	INTB	Not Connected
21	GPA0	CH1_EN1
22	GPA1	CH2_EN1
23	GPA2	CH3_EN1
24	GPA3	CH4_EN1
25	GPA4	CH1_EN2
26	GPA5	CH2_EN2
27	GPA6	CH3_EN2
28	GPA7	CH4_EN2

8.2.2 Port Expander D

Physical Pin	Pin Name	Connected Signal
1	GPB0	CS_ADC
2	GPB1	Not Connected
3	GPB2	DAC1_SYNC
4	GPB3	DAC2_SYNC
5	GPB4	DAC3_SYNC
6	GPB5	DAC4_SYNC
7	GPB6	DAC5_SYNC
8	GPB7	DAC6_SYNC
9	V_{DD}	V3
10	V_{SS}	DGND
11	$\overline{CS}$	CS_SPD
12	SCK	SCLK
13	SI	DIN
14	SO	DOUTD
15	A0	VCC
16	A1	VCC
17	A2	DGND
18	$\overline{RESET}$	V3
19	INTA	Not Connected
20	INTB	Not Connected
21	GPA0	EN_MUX1
22	GPA1	EN_MUX2
23	GPA2	EN_MUX3
24	GPA3	EN_MUX4
25	GPA4	MUX_0
26	GPA5	MUX_1
27	GPA6	MUX_2
28	GPA7	MUX_3

8.3 DAC1 connections

Physical Pin	Pin Name	Connected Signal
1	SYNC	DAC1_SYNC
2	AV_{DD}	V3
3	$V_{OUT}A$	CH1_CTR1
4	$V_{OUT}C$	CH3_CTR1
5	$V_{OUT}E$	CH1_CTR2
6	$V_{OUT}G$	CH3_CTR2
7	$V_{REF}IN/V_{REF}OUT$	Not Connected
8	$V_{OUT}H$	CH4_CTR2
9	$V_{OUT}F$	CH2_CTR2
10	$V_{OUT}D$	CH4_CTR1
11	$V_{OUT}B$	CH2_CTR1
12	GND	DGND
13	D_{IN}	DIN
14	SCLK	SCLK

Note: The other DACs aren't shown, because they have the same kind of connections, only differing in the channel's numbers.

8.4 MUX1 Connections

Physical Pin	Pin Name	Connected Signal
1	VDD	VP12
2	NC	Not Connected
3	NC	Not Connected
4	S16	RD4_CH4
5	S15	RD3_CH4
6	S14	RD2_CH4
7	S13	RD1_CH4
8	S12	RD4_CH3
9	S11	RD3_CH3
10	S10	RD2_CH3
11	S9	RD1_CH3
12	GND	AGND
13	NC	Not Connected
14	A3	MUX_3
15	A2	MUX_2
16	A1	MUX_1
17	A0	MUX_0
18	EN	EN_MUX1
19	S1	RD1_CH1
20	S2	RD2_CH1
21	S3	RD3_CH1
22	S4	RD4_CH1
23	S5	RD1_CH2
24	S6	RD2_CH2
25	S7	RD3_CH2
26	S8	RD4_CH2
27	VSS	VN12
28	D	OUT_MUXs

Note: The other MUXs aren't shown, because they have the same kind of connections only differing in the channel's numbers, although, MUX2 and MUX4 do have some unused pins that were connected to the analogue ground AGND.

8.5 Mapping of the PMTs channels

PMT Number	Channel Name	Port Expander for enable HV	MUX Number	MUX address code	DAC Number	DAC Channel
1	CH1_1	A	1	0000	1	A
2	CH2_1		1	0001	1	B
3	CH3_1		1	0010	1	C
4	CH4_1		1	0011	1	D
5	CH1_2		1	0100	1	E
6	CH2_2		1	0101	1	F
7	CH3_2		1	0110	1	G
8	CH4_2		1	0111	1	H
9	CH1_3		1	1000	2	A
10	CH2_3		1	1001	2	B
11	CH3_3		1	1010	2	C
12	CH4_3		1	1011	2	D
13	CH1_4		1	1100	2	E
14	CH2_4		1	1101	2	F
15	CH3_4		1	1110	2	G
16	CH4_4		1	1111	2	H
17	CH1_5	B	2	0000	3	A
18	CH2_5		2	0001	3	B
19	CH3_5		2	0010	3	C
20	CH4_5		2	0011	3	D
21	CH1_6		2	0100	3	E
22	CH2_6		2	0101	3	F
23	CH3_6		2	0110	3	G
24	CH4_6		2	0111	3	H
25	CH1_7		3	0000	4	A
26	CH2_7		3	0001	4	B
27	CH3_7		3	0010	4	C
28	CH4_7		3	0011	4	D
29	CH1_8		3	0100	4	E
30	CH2_8		3	0101	4	F
31	CH3_8		3	0110	4	G
32	CH4_8		3	0111	4	H
33	CH1_9	C	3	1000	5	A
34	CH2_9		3	1001	5	B
35	CH3_9		3	1010	5	C
36	CH4_9		3	1011	5	D
37	CH1_10		3	1100	5	E
38	CH2_10		3	1101	5	F
39	CH3_10		3	1110	5	G
40	CH4_10		3	1111	5	H

41	CH1_11		4	0000	6	A
42	CH2_11		4	0001	6	B
43	CH3_11		4	0010	6	C
44	CH4_11		4	0011	6	D
45	CH1_12		4	0100	6	E
46	CH2_12		4	0101	6	F
47	CH3_12		4	0110	6	G
48	CH4_12		4	0111	6	H

8.6 Code with the implementation of the HVREMOTE class

```
#!/usr/bin/python
"""------------------------------------------------------------------------
------------
                                hvRemote.py
Description:    HV remote class
Class:          HVREMOTE
------------------------------------------------------------------------
---------"""
# HV Remote
# Contains 3 port expanders one DAC and one ADC
#
from RPiHVREMOTE.MCP23S17_v3 import MCP23S17
from RPiHVREMOTE.DAC7568_v2 import DAC7568
from RPiHVREMOTE.MAX1240_v2 import MAX1240
import spidev
import time
import RPi.GPIO as GPIO
from RPiHVREMOTE import map

class HVREMOTE(object):
    # chip select pin - GPIO board numeration
    # to add more
    CHIP_SELECT_SPA=11#15
    CHIP_SELECT_SPB=11#13
    CHIP_SELECT_SPC=11
    CHIP_SELECT_SPD=11#12
    CS_ADC = 1
    DAC1_SYNC = 3
    DAC2_SYNC = 4
    DAC3_SYNC = 5
    DAC4_SYNC = 6
    DAC5_SYNC = 7
    DAC6_SYNC = 8

    # Addresses of the port expanders
    PORT_EXPANDER_ADD_A = 0b000 #0b000
    PORT_EXPANDER_ADD_B = 0b001 #0b001
    PORT_EXPANDER_ADD_C = 0b011 #0b111
    PORT_EXPANDER_ADD_D = 0b111 #0b011

    def __init__(self, spiDevice=spidev.SpiDev(), iGPIO = GPIO, board_id
= 0, chip_select_hvRemote = 40):
        self.__spi = spiDevice
        self.__board_id = board_id
        self.__chip_select_hvRemote = chip_select_hvRemote
```

```
        self.__GPIO = iGPIO

        #initalization of the port expanders, dacs and adc
        self.__mcp1 = MCP23S17(self.__spi, self.PORT_EXPANDER_ADD_A, -
1, self.CHIP_SELECT_SPA, self.__GPIO)
        self.__mcp2 = MCP23S17(self.__spi, self.PORT_EXPANDER_ADD_B, -
1, self.CHIP_SELECT_SPB, self.__GPIO)
        self.__mcp3 = MCP23S17(self.__spi, self.PORT_EXPANDER_ADD_C, -
1, self.CHIP_SELECT_SPC, self.__GPIO)
        self.__mcp4 = MCP23S17(self.__spi, self.PORT_EXPANDER_ADD_D, -
1, self.CHIP_SELECT_SPD, self.__GPIO)
        self.__dac1 = DAC7568(self.__spi, self.DAC1_SYNC, self.__mcp4)
        self.__dac1.enableReference('flexible', 2)
        self.__dac2 = DAC7568(self.__spi, self.DAC2_SYNC, self.__mcp4)
        self.__dac3 = DAC7568(self.__spi, self.DAC3_SYNC, self.__mcp4)
        self.__dac4 = DAC7568(self.__spi, self.DAC4_SYNC, self.__mcp4)
        self.__dac5 = DAC7568(self.__spi, self.DAC5_SYNC, self.__mcp4)
        self.__dac6 = DAC7568(self.__spi, self.DAC6_SYNC, self.__mcp4)
        self.__adc = MAX1240(self.__spi, -1, self.CS_ADC, self.__mcp4)
        self.__GPIO.setup(self.__chip_select_hvRemote, self.__GPIO.OUT)


        self.enableDisableAllHvChannels(False)


    def enableDisableAllHvChannels(self, onOff):
        """
        : onOff - True or False
        disable or enable all the HV channels corresponding to port expan
der A, B, C
        """
        self.__GPIO.output(self.__chip_select_hvRemote, self.__GPIO.HIGH)
        if onOff == False:
            command = 0x00
        else:
            command = 0xFFFF
        self.__mcp1.writeGPIO(command) # from drawing we will have these
3 for enable/disable HV - note here we do not care about pmt number
        self.__mcp2.writeGPIO(command)
        self.__mcp3.writeGPIO(command)
        self.__GPIO.output(self.__chip_select_hvRemote, self.__GPIO.LOW)


    def enableDisableChannel(self, hvChannelNumber, onOff):
        """
        : hvChannelNumber - int value - from 1 to 48 **Respect the PMT nu
meration**
        : onOff - bool value - True or False
        Enable or disable the output voltage for one HV channel
```

```
        """
        assert hvChannelNumber in range(1, 49)
        self.__GPIO.output(self.__chip_select_hvRemote, self.__GPIO.HIGH)
     #just to make sure that the correct one is enabled
        #finds which channels correspond to what
        signalName = map.pmtMap2channelSignal[hvChannelNumber]
        expanderGpio = map.channel2chip[signalName[0]]
        if expanderGpio[0] == 1:
            self.__mcp1.digitalWrite(expanderGpio[1], onOff)
        elif expanderGpio[0] == 2:
            self.__mcp2.digitalWrite(expanderGpio[1], onOff)
        else:
            self.__mcp3.digitalWrite(expanderGpio[1], onOff)

        self.__GPIO.output(self.__chip_select_hvRemote, self.__GPIO.LOW)
        return self.__mcp4.readGPIO()


    def readHVChannel(self, hvChannelNumber):
        """
        : hvChannelNumber - int value - from 1 to 48 **Respect the PMT nu
meration**
        Enable of the ADC
            """
        assert hvChannelNumber in range(1, 49)
        self.__GPIO.output(self.__chip_select_hvRemote, self.__GPIO.HIGH)
        signalName = map.pmtMap2channelSignal[hvChannelNumber]
        chipSelection = map.channel2chip[signalName[0]] #chipSelection[3]
 tells what MUX to use and chipSelection[4] is the word for the selectors
        GPIOA = chipSelection[4] << 4 | chipSelection[3] #MCP4's GPIOA ha
s the enables of the MUXs and its selectors
        GPIOB = 0b11111101
        GPIO_word = (GPIOB << 8) | GPIOA
        self.__mcp4.writeGPIO(GPIO_word)
        value = self.__adc.readVoltage()
        #read/communicate with adc ...execute 5 readings
            #@ that to be added
        # disables all the outputs
        GPIOA = 0x00
        GPIO_word = (GPIOB << 8) | GPIOA
        self.__mcp4.writeGPIO(GPIO_word)                                # n
ot sure if it will be required - to be tested with the chips
        self.__GPIO.output(self.__chip_select_hvRemote, self.__GPIO.LOW)
        return value

    #Set the HV of PMT
    # @ hvChannelNumber - from 1 up to 48 -- respect the PMT numeration
    # @ hvOrder - float value of the requested HV // add restrisctions
    def setHV(self, hvChannelNumber, hvOrder):
```

```
        """
        : hvChannelNumber - int value - from 1 to 48 **Respect the PMT nu
meration**
        : hvOrder - float value of the requested HV // TODO add restrisct
ions
        Set the HV of PMT
        """
        assert hvChannelNumber in range(1, 49)
        self.__GPIO.output(self.__chip_select_hvRemote, self.__GPIO.HIGH)
        convertedValue = hvOrder #conversion of the hv value
        signalName = map.pmtMap2channelSignal[hvChannelNumber]
        dacSelection = map.channel2chip[signalName[0]]
        dacNumber = dacSelection[5]
        dacChannel = dacSelection[6]
        if dacNumber == 1:
            self.__dac1.writeAndUpdateChannel(dacChannel, convertedValue)
        elif dacNumber == 2:
            self.__dac2.writeAndUpdateChannel(dacChannel, convertedValue)
        elif dacNumber == 3:
            self.__dac3.writeAndUpdateChannel(dacChannel, convertedValue)
        elif dacNumber == 4:
            self.__dac4.writeAndUpdateChannel(dacChannel, convertedValue)
        elif dacNumber == 5:
            self.__dac5.writeAndUpdateChannel(dacChannel, convertedValue)
        else:
            self.__dac6.writeAndUpdateChannel(dacChannel, convertedValue)

        self.__GPIO.output(self.__chip_select_hvRemote, self.__GPIO.LOW)

    def set_spiSpeed(self, speed):
        self.__spi.max_speed_hz = speed

    def reset(self):
        self.__dac1.reset()
        self.__dac2.reset()
        self.__dac3.reset()
        self.__dac4.reset()
        self.__dac5.reset()
        self.__dac6.reset()
        self.__mcp1.reset()
        self.__mcp2.reset()
        self.__mcp3.reset()
        self.__mcp4.reset()
```

8.7 Code of the GUI

```
import sys
import spidev
import RPi.GPIO as GPIO
import time
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
import pyqtgraph as pg
import pyqtgraph.exporters
import numpy as np
from RPiHVREMOTE.hvRemote import HVREMOTE

##### OPENING AND CONFIGURATION OF SPI INTERFACE AND RPI GIPOs #####
spi = spidev.SpiDev()
spi.open(0, 0)
spi.max_speed_hz = 976000
spi.no_cs = True
GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD)

##### INITIALIZATION OF HVREMOTE BOARDS #####
board1 = HVREMOTE(spi, GPIO, 0x00, 40)

class PmtChannelPlot():

    idCounter = 1
    def __init__(self, *args, **kwargs):

        self.__id = PmtChannelPlot.idCounter
        self.__plot = pg.plot()
        PmtChannelPlot.idCounter += 1



class Channel(QHBoxLayout):

    idCounter = 1
    def __init__(self, *args, **kwargs):
        super(Channel, self).__init__(*args, **kwargs)

        self.state = False
        self.__id = Channel.idCounter
        Channel.idCounter += 1

        self.__checkBox = QCheckBox()
```

```
        self.__checkBox.setCheckState(Qt.Unchecked)
        self.__checkBox.stateChanged.connect(self.change_state)
        self.__channelName = QLabel("Channel " + str(self.__id))
        self.__channelName.setAlignment(Qt.AlignHCenter | Qt.AlignVCenter
)

        self.addWidget(self.__checkBox)
        self.addWidget(self.__channelName)
        self.setSpacing(5)

    def change_state(self):
        if self.state == False:
            self.state = True
            board1.enableDisableChannel(self.__id, True)
            print("Channel " + str(self.__id) + " Enabled")
        else:
            self.state = False
            board1.enableDisableChannel(self.__id, False)
            print("Channel " + str(self.__id) + " Disabled")

    def check_box(self):
        self.__checkBox.blockSignals(True)
        self.__checkBox.setCheckState(Qt.Checked)
        self.state = True
        self.__checkBox.blockSignals(False)

    def uncheck_box(self):
        self.__checkBox.blockSignals(True)
        self.__checkBox.setCheckState(Qt.Unchecked)
        self.state = False
        self.__checkBox.blockSignals(False)


class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)
        #*args is a non-keyworded variable length argument list
        #**kwargs is a keyworded variable length arguments

        self.setWindowTitle("HVRemote Tester")
        self.setGeometry(400, 400, 640, 480)
        self.__chList = []
        for n in range(1, 49):
            newChannel = Channel()
            self.__chList.append(newChannel)

        self.__setHVTab = QWidget()
        self.__readTab = QWidget()
```

```
        self.setHVTab_config()
        self.readTab_config()

        tabs = QTabWidget()
        tabs.addTab(self.__setHVTab, "Set HV and Enable/Disable")
        tabs.addTab(self.__readTab, "Reading of PMT Channels")

        self.setCentralWidget(tabs)


    def setHVTab_config(self):

        page_layout = QVBoxLayout()
        enableDisable_layout = QGridLayout()
        enableDisableAll_layout = QHBoxLayout()
        write_layout = QHBoxLayout()


        ##### BUILDING THE ENABLE/DISABLE INTERFACE OF THE CHANNELS #####
        for i in range(0, 6):
            for j in range(0, 8):
                enableDisable_layout.addLayout(self.__chList[j + i * 8],
i, j)
        enableDisable_layout.setContentsMargins(50, 25, 50, 50)
        enableDisable_layout.setSpacing(40)


        ##### BUTTON TO ENABLE/DISABLE ALL CHANNELS #####
        enableAll_btn = QPushButton("Enable All")
        enableAll_btn.clicked.connect(self.enableAll_clicked)
        disableAll_btn = QPushButton("Disable All")
        disableAll_btn.clicked.connect(self.disableAll_clicked)

        enableDisableAll_layout.addWidget(enableAll_btn)
        enableDisableAll_layout.addWidget(disableAll_btn)
        enableDisableAll_layout.setAlignment(Qt.AlignCenter)
        enableDisableAll_layout.setSpacing(100)
        enableDisableAll_layout.setContentsMargins(0, 25, 0, 25)


        ##### BOX TO WRITE THE DESIRED VOLTAGE AND COMBO BOX TO SELECT TH
E PMT WE WANT TO MODIFY #####
        pmtSelector = QComboBox()
        for n in range(1, 49):
            pmtSelector.addItem(str(n))
        pmtSelector_label = QLabel()
        pmtSelector_label.setText("PMT Number:")
        pmtSelector_label.setAlignment(Qt.AlignRight | Qt.AlignVCenter)
        hvValue = QSpinBox() #QDoubleSpinBox()
```

```
        hvValue.setMaximum(4095)
        hvValue_label = QLabel()
        hvValue_label.setText("Voltage in V:")
        hvValue_label.setAlignment(Qt.AlignRight | Qt.AlignVCenter)
        writeButton = QPushButton("Set Voltage")
        writeButton.clicked.connect(lambda: self.writeButton_clicked(int(
pmtSelector.currentText()), hvValue.value()))

        write_layout.addWidget(pmtSelector_label)
        write_layout.addWidget(pmtSelector)
        write_layout.addWidget(hvValue_label)
        write_layout.addWidget(hvValue)
        write_layout.addWidget(writeButton)
        write_layout.setAlignment(Qt.AlignCenter)
        write_layout.setContentsMargins(0, 0, 0, 50)
        write_layout.setSpacing(25)


        ##### BUILDING THE PAGE OF THE INTERFACE WITH ALL THE PREVIOUS LA
YOUTS #####
        page_layout.addLayout(enableDisableAll_layout)
        page_layout.addLayout(enableDisable_layout)
        page_layout.addLayout(write_layout)

        self.__setHVTab.setLayout(page_layout)

    def readTab_config(self):
        plots_layout = QVBoxLayout()
        read_layout = QHBoxLayout()

        ##### COMBO BOX TO SELECT THE PMT NUMBER AND BUTTON TO READ THE C
HOSEN CHANNEL #####
        pmtSelector = QComboBox()
        for n in range(1, 49):
            pmtSelector.addItem(str(n))
        pmtSelector_label = QLabel()
        pmtSelector_label.setText("PMT Number:")
        pmtSelector_label.setAlignment(Qt.AlignRight | Qt.AlignVCenter)
        readButton = QPushButton("Read Voltage")
        readButton.clicked.connect(lambda: self.readButton_clicked(int(pm
tSelector.currentText())))
        label = QLabel()
        label.setText("Voltage in V:")
        label.setAlignment(Qt.AlignRight | Qt.AlignVCenter)
        self.read_lcd = QLCDNumber()

        read_layout.addWidget(pmtSelector_label)
        read_layout.addWidget(pmtSelector)
        read_layout.addWidget(readButton)
```

```
        read_layout.addWidget(label)
        read_layout.addWidget(self.read_lcd)
        read_layout.setAlignment(Qt.AlignCenter)
        read_layout.setContentsMargins(0, 0, 0, 50)
        read_layout.setSpacing(25)

        plots_layout.addLayout(read_layout)
        self.__readTab.setLayout(plots_layout)

    def enableAll_clicked(self):
        for ch in self.__chList:
            ch.check_box()

        board1.enableDisableAllHvChannels(True)
        print("All channels Enabled")

    def disableAll_clicked(self):
        for ch in self.__chList:
            ch.uncheck_box()

        board1.enableDisableAllHvChannels(False)
        print("All channels Disabled")

    def writeButton_clicked(self, chNumber, hvValue):
        board1.setHV(chNumber, hvValue)
        print("Channel " + str(chNumber) + " set to: " + str(hvValue))

    def readButton_clicked(self, chNumber):
        read_value = board1.readHVChannel(chNumber)
        self.read_lcd.display(read_value)
        print("Channel " + str(chNumber) + " reading: " + str(read_value)
)
        return read_value



app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec_()
board1.reset()
spi.close()
```

8.8 Code for testing the HVREMOTE class

```
#!/usr/bin/python
from RPiHVREMOTE.hvRemote import HVREMOTE
import spidev
import time
import RPi.GPIO as GPIO
import matplotlib.pyplot as plt
import numpy as np

##### BOARD IDs #####
BOARD1_ID = 0b000

##### BOARD CS #####
BOARD1_CS = 40

##### OPEN SPI LINE #####
spi = spidev.SpiDev()
spi.open(0, 0)
spi.max_speed_hz = 976000
spi.no_cs = True

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD)

board1 = HVREMOTE(spi, GPIO, BOARD1_ID, BOARD1_CS)

##### TESTS #####
try:

    #for speed in [976000, 244000, 61000, 15200]:
    for channel in range(1, 9):
    #channel = 1
        name1 = "writeReadCH" + str(channel)
        name2 = "errorCH" + str(channel)
        file_name1 = name1 + ".txt"
        file_name2 = name2 + ".txt"
        test_file1 = open(file_name1, 'w')
        test_file2 = open(file_name2, 'w')

        speed = 1953000
        board1.set_spiSpeed(speed)

        i = 0
        while i < 4096:
            board1.setHV(channel, i)
            time.sleep(0.001)
```

```
            value_read = board1.readHVChannel(channel)
            time.sleep(0.001)

            test_file1.write(str(i))
            test_file1.write("\t")
            test_file1.write(str(value_read))
            test_file1.write("\n")

            error = i - value_read
            test_file2.write(str(i))
            test_file2.write("\t")
            test_file2.write(str(error))
            test_file2.write("\n")

            i = i + 15

        test_file1.close()
        test_file2.close()

    board1.reset()
    spi.close()

except KeyboardInterrupt:
    board1.reset()
    spi.close()
```